# Out of Sight, Out of Mind:

# A Comprehensive Study on the Prevalence and Security Impact of Orphaned Web Pages

S.R.G. Pletinckx

Master Thesis

# Out of Sight, Out of Mind:

A Comprehensive Study on the Prevalence and Security Impact of Orphaned Web Pages

by

S.R.G. Pletinckx

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday July 22, 2021 at 10:00 AM.

| | | |
|---|---|---|
| Student number: | 4459687 | |
| Project duration: | November 9, 2020 – July 22, 2021 | |
| Thesis committee: | Dr. Z. Erkin, | TU Delft, chair |
| | Dr. S. Picek, | TU Delft |
| | Dr.-Ing T. Fiebig, | TU Delft, supervisor |

TU Delft

# Abstract

Security misconfigurations and neglected updates commonly lead to systems being vulnerable. Ranging from default passwords to unpatched software, many systems, such as websites or databases, are being compromised due to these pitfalls. Often stemming from human error, it is difficult to avoid these misconfigurations, which is why they are repeatedly seen in the wild. Especially in the context of websites, we often find pages that were *forgotten*, that is, they were left online after they served their purpose and were never updated thereafter.

In this thesis, we introduce the first methodology to detect such forgotten or *orphaned* web pages, a type of misconfiguration that has seen little attention. We combine the historic data set of the Internet Archive with active measurements to identify pages that can no longer be reached via a path from the index page, yet remain accessible through their specific URL. We show the efficacy of our approach and the real-world relevance of the issue of orphaned web pages by applying it to a sample of 100,000 domains from the Tranco Top 1M. This particular type of misconfiguration can pose a serious threat, as they can lead to forgotten and unmaintained web pages, which may not have seen security updates.

In our measurements, we find 1,953 pages on 907 unique domains that are orphaned, some of which are 20 years old. In our subsequent security analysis, we find that these pages are significantly ($p < 0.01$ using $\chi^2$) more likely to contain some vulnerabilities than maintained pages: 7.1% of orphaned pages suffer from simple XSS vulnerabilities, while only 0.9% of maintained pages are vulnerable against this type of attack. We encounter similar patterns for following best security practices, such as for the use of Content Security Policies (CSPs) and setting HTTP Security Headers. To allow researchers to reproduce our results and practitioners to scrutinize their own pages, we provide an implementation of our methodology as open source software.

# Preface

I often think that the worst part of master theses are the prefaces and the LinkedIn posts written upon finalizing the process. It is not too uncommon to encounter sentences such as "*what a journey it has been*" or "*I never expected at the beginning of this thesis to get here*", addressing some sort of humble bewilderment regarding the status quo one just achieved. I am concealed from this bewilderment, not because I was not challenged during the process, nor because I had any shortage of difficulties doing the research. I certainly had my share of struggle. The reason I finished the job without surprise is because it was always the plan to finish the job. I am clueless as to why one would embark on such a project with the mindset of "Well, here goes nothing..." only to pursue a moment of astonishment later on when the project is indeed completed. If anything, doing the thesis project was the challenge I was looking most forward to when enrolling for the master program. Boasting about it on social media therefore feels a bit off, not to mention the cringe-worthy pictures, accompanied with a quote or text as if one just became a noble laureate in their field. This does not take away the pride I feel for the work accomplished, although I am perhaps more proud of what the work represents. The world of science is namely one of the only places where a true form of apprenticeship is still in place. The most notorious and straightforward example being the doctoral progression to obtain the title of PhD. It is aimed to test and train one's scientific aptitude, therefore preventing as much as possible the introduction of charlatans into the world of science. Whether successful or not is a different dialectic, but it is one of many walls and defense mechanisms science has in place to safeguard the integrity and relentless skepticism necessary for finding the truths and details of this world. Having been part of a microcosmic version of this apprenticeship is therefore an extreme privilege, one which I will never take for granted.

Please allow me to thank a few people who were imperative to this process. The first being my supervisor, Tobias Fiebig, for his time and effort in guiding me through this research. I am forever in dept for the early insights you gave me into academia, ranging from introducing me to influential people in the field, to involving me in the review process of your conference duties. I hope this thesis can furnish as the start, and not the end, of our scientific collaboration. Second, many thanks go to Kevin Borgolte for giving clever insights into this study and providing a plethora of valuable feedback on my writing. Kevin is also the living proof that general practitioners are not the only doctors with 'hard-to-decipher' handwriting ;-). Furthermore, I would like to thank Zeki Erkin for being my responsible professor, as well as the chair of the thesis committee. I had to ask you and Tobias many times for an autograph, therefore showing that computer scientists are in fact the rockstars of our time –at least if it was up to the bureaucratic formalities. Also thank you to Stjepan Picek for being a member of the thesis committee. Your lightning-speed replies to my emails were always greatly appreciated.

One should never shy away from a moment to express gratitude, and because I reached the end of my studies I cannot help but to reflect on some of the people that were important to me during this –very well then– *journey*. I will allow myself the freedom to thus also thank them in this preface. (After all it is my preface.) I am forever grateful to Christian Doerr for showing me the first steps into the world of cyber security, specifically the scientific part of it. Our meetings often taught me much more than 3 weeks of course work in other subjects, and if it was not for you my skills and drive would be far less than what they are today. I hope our paths cross again in the future. I would also like to thank Rolf van Wegberg for spending extra time with me on the project I did for his course. I very much enjoyed working together, and I learned a lot from our collaboration. Also thank you to my friends, both in the Netherlands as well as in Belgium, for, well, being my friends. In particular the 'IDEA League' group in Delft, Giulio, Johannes, and José, for showing that hard work and good times can perfectly be combined. The Spartan Workout group, specifically Bart van der Laar, for teaching me mental toughness and an attitude to never back down. Also my friends Barry, Broes, Gaetan, Phili, and Radu for always welcoming me back in Belgium as if I never left.

Thank you to my older brother, Stefano, for being an inspiration ever since I can remember. The things you achieve never fail to impress me, and I still learn a lot from having you as my brother. Thank you to Dennis, for teaching me the first steps into computer science, and encouraging me to go and pursue a degree at TU Delft. If it was not for you, I would have maybe never discovered this passion, and potentially still be lost in life. Perhaps most importantly, I want to thank my parents, for their unconditional support in my studies. Thank you to my father, for at one point working two jobs to finance my degree, and to my mother, for always welcoming me back with a smile, even when I came back only after saying for the 20th time "*I am sure that next weekend I will stop by.*" There is no conceivable way in which I can ever repay you both for giving me the opportunity to study. Finally, I want to thank my partner Irene, for being my Spanish sun in this awfully dreadful Dutch weather, and for taking care of things around when time was scarce.

*S.R.G. Pletinckx*
*Delft, July 2021*

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Listings

# 1 Introduction

The World Wide Web is an ever-changing landscape. Websites today have an exceedingly different look and feel compared to websites of 30 years ago when The Web was initially invented[1], both from the outside as well as the inside, that is, the code that constructs a website. The fundamental concept, however, of how The Web functions remained more or less the same over the years. It's main goal is to efficiently locate, access, and share information stored on different computer systems. With this endeavor, a "web" of documents would emerge, which are all connected to, and accessible via, the Internet.

Informally speaking, The Web works as follows: A website consists of multiple web pages, and each web page can be identified by a unique Uniform Resource Locator (URL). URLs simplify navigation and access on The Web as they store the location within a computer network (such as the Internet) of their respective resource, which can be a web page, image, video, or any other type of file. Within web pages, it is possible to reference other web pages using hyperlinks. When clicked, these hyperlinks make use of a page's URL to identify and navigate toward the location of the referenced web resource.

Also from a producer and consumer perspective, we have seen The Web change over the years. Where The Web was initially used solely by tech-savvy people, today it has become trivial to acquire your personal domain and deploy a website (the producers). As for usage of The Web (the consumers), it has become inconceivable to image our day-to-day lifestyle and habits without the presence of the Internet, and in particular The Web. This ease-of-access has enabled wonderful possibilities and great enhancements to society in the form of widespread access to knowledge, freedom of speech, as well as opportunity to interact with and learn from other cultures across the globe. Unfortunately, The Web also carries its demons. Despite its fruitful contribution to the advancement of society, The Web also introduced harmful implications of its usage such as Internet addiction, privacy violations, and cybercrime.

Although technical details such as DNS configurations, packet broadcasting, and even web-programming can be automated and outsourced, making a website secure is not as straightforward as making a website in itself. It requires not only a sound knowledge of the underlying programming language(s) behind your web service, but also a thorough and up-to-date understanding of the current vulnerabilities and exploits on web application systems, such as for example compiled in the OWASP Top Ten [21]. However, since more and more web application technologies are maintained by tech companies or open source projects, a good first step is to just keep the technologies of your web application up-to-date.

Despite the fact that keeping your website up-to-date is imperative for ensuring that it is free of bugs and vulnerabilities, it has been shown that this is a cumbersome endeavor that is rarely achieved fully and often not at all [36]. Common causes are delaying or ignoring critical security updates [37], keeping default configurations [20], and other security misconfigurations, often rooted in human error [57, 18].

A recent example of a security misconfiguration that led to a compromise is Deloitte's "Test your Hacker IQ" campaign [11]. This advertisement campaign ran in 2015 and made use of a promotional website that remained online long after the actual campaign ended. In November 2020, an IT consultant managed to find the old domain, and was able to retrieve the access credentials to the MySQL database. Presumably, the website was forgotten by the organization, which left it unmaintained and prone to vulnerabilities. Although

[1]As with many great inventions, there is some debate on when and by whom The Web was invented. Generally, its invention is attributed to Sir Tim Berners-Lee in 1989.

these vulnerabilities might not have been known at the time of deploying the website, neglecting maintenance of the website (such as ignoring security updates) avoids protecting the domain against exploits much later discovered than the initial deployment date.

The Deloitte example highlights that website administrators can lose track of the state of their website, which in this case led to an old and unmaintained domain exposing data for years. Finding such no-longer-used *domains* is feasible by, for example, tracking Certificate Transparency (CT) logs to identify hosts that stop renewing their certificates but remain reachable, or by using passive DNS traces to identify domains that, over time, receive significantly less traffic. However, these techniques do not allow to identify URLs of abandoned content on a *single* domain but at a different path, like a discontinued web application hosted at `example.com/web-app` that receives no longer updates, while `example.com` remains actively maintained and used. In this thesis, we aim to shine light on this blind spot and we develop a methodology to identify orphaned URLs of single domains in the wild and at-scale.

## 1.1. Orphan Pages Primer

To further introduce the topic of this work, it is required to have a rudimentary understanding about what is meant by an orphaned web page. We express what it means for web pages (and the URL pointing to them) to be orphaned in terms of who knows about them (administrators vs. the public). This leads to four quadrants (Figure 1.1) that, much like a Johari window [39], characterize the pages of a website as known and unknown to the administrator and the public.

Most websites have pages that are intended for, and known to, the public. At the same time, some pages are usually not known to the public or even intended to be accessed by the public, such as administrative interfaces, content management, but also internal pages, like a company's intranet. In general, administrators should "know" about both types of pages. However, they may become unaware of them and then these pages become orphaned. Eventually, these pages will be unmaintained, and, for example, may not receive security updates that address existing vulnerabilities. In our work, we identify these orphaned pages, with a specific focus on Quadrant 1 of Figure 1.1, that is, pages that were once intended for public access but have since been forgotten by the administrators.

One way how a page can become forgotten and unmaintained is by a misconfiguration during the removal of a web page. Successful removal of a web page requires two steps: first, removing the page from the webserver, and, second, making sure no other page on the website links to the removed page. If merely the latter step is performed, then the page only appears to be removed because it is no longer accessible by navigating on the website (i.e., the page is no longer reachable by following links on the website). However, since the page has not actually been removed from the webserver, it remains accessible by navigating to the page's URL directly. We classify these pages as "orphaned" web pages (see Chapter 2 for a formal definition and background on orphaned web pages).

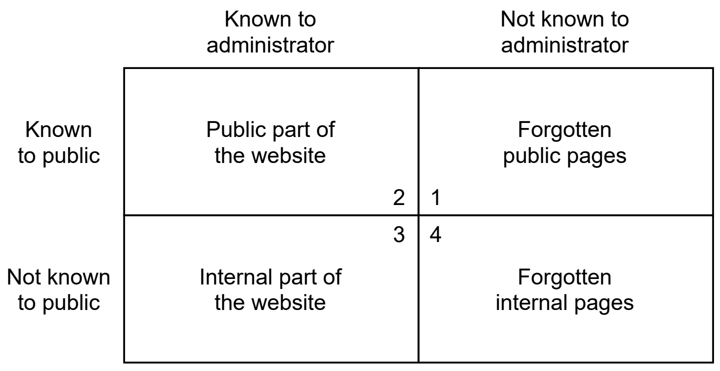


Figure 1.1: Matrix of known and unknown pages on a website. Columns differentiate between pages that are known, and pages that are not known by the administrator (called "administrator" for simplicity, this can be a team). Rows differentiate between the public's knowledge.

## 1.2. Posing the Right Questions

The objective of this work is to provide the first comprehensive analysis of orphaned web pages in the wild and at Internet-scale. To do so, we pose the following research question:

**(RQ):** What is the prevalence and security impact of orphaned web pages?

Naturally, this question divides itself into two parts: the prevalence and the security impact. To study the prevalence, we require a large-scale network measurement study that can quantify the presence of orphan pages on the Internet. This can answer the following question:

**(RQ1) Prevalence:** What is the prevalence of orphaned web pages in the wild?

Inevitably, since no systematic method has been designed yet to detect such pages, we must first develop a technique to perform these measurements. Doing so is no sinecure, and hence requires a structured research approach to achieve adequate and scientifically sound results. We therefore divide **(RQ1)** into the following sub-questions:

**(RQ1.1):** How do we define and characterize orphaned web pages?
**(RQ1.2):** How to systematically detect orphaned web pages in the wild and at Internet-scale?
**(RQ1.3):** What is a lower bound on the number of orphaned web pages in the wild?

To study the security impact of orphaned web pages we need to analyze how susceptible these pages are to (common) vulnerabilities. We pose the following question:

**(RQ2) Security Impact:** What is the security impact of orphaned web pages?

Which can again be divided into sub-questions:

**(RQ2.1):** How do orphan pages look like from a programming perspective?
**(RQ2.2):** How many orphan pages are susceptible to a Cross-Site Scripting vulnerability?
**(RQ2.3):** How many orphan pages are susceptible to an SQL Injection?
**(RQ2.4):** How does a vulnerability on an orphan page differ from a vulnerability on a non-orphan page when it comes to the security impact it can have on a website?

## 1.3. Finding the Right Answers

As became apparent from our research questions, it is crucial to develop a sound methodology to detect these forgotten orphaned web pages in the wild and, preferably, at-scale. The latter requirement makes this work challenging. Key to the nature of an orphaned web page is the removal of identifiers or pointers (such as URLs) to the page. As a result, we are essentially left with no information on how or where to identify these pages. This suggests the need for guesswork and brute-force enumeration to find URLs of orphaned web pages. This is unfortunately not scalable, let alone effective, and hence asks for a better approach to solve this problem.

Fortunately, the availability of web archives can help us in this. Web archives perform systematic crawls of pages on the Internet, and store their content and structure to create a historic overview of websites on the World Wide Web. By consulting historic versions of a website, we can get a grasp of which pages were accessible on a domain at some point in time. Comparing this data to the current sitemap of a domain allows us to extract previously active pages that appear unlisted today. This gives a more tailored list of potential orphan candidates –on a per-domain basis– while simultaneously eliminating the need for a brute-force approach.

Once a list of potential orphan pages is constructed, we can probe each individual page to see whether or not some of them are still online. Doing so can again cause issues if we want to perform this step at Internet-scale due to large target lists and the potential of non-orphaned web pages being unlisted on the archive. As such, we will develop filters and heuristics that can eliminate some of the candidate URLs. These filters and heuristics will be based on file extensions (by eliminating non-webpages such as PDF or XLS files), and the structure of the URL (by detecting dynamically generated URLs based on common prefix structures). Due to the novelty of this work, many of the filters and heuristics need to be developed (and hence evaluated) exclusively for this research.

After detecting unlisted web pages that are still online and responsive, we need to evaluate their orphan status. We do so by comparing each page to its version in the web archive and determining the similarity among the two. The more similar a web page is to its archived version, the more we assume the page has been left unmaintained, at least for the time since the last archive crawl. Our methodology for this is based on Google's *simhash* which creates a fingerprint for each page that can be effectively constructed and compared to other fingerprints, again at-scale. Together with the similarity as well as the age of a page, we calculate an orphan score that we use to distinguish a group of likely orphaned web pages. We develop this methodology into a modular open source implementations and deploy it on 20 servers within the university network. To evaluate our methodology, we perform a month-long measurement study by running the implementation on a subset of 100,000 domains taken from the Tranco Top 1M [33]. After classifying the pages, we can answer **(RQ1)** and proceed to the security analysis of our orphan pages.

Important to stating the security status of the found orphan pages is putting the results into context. That is, we need a control group of non-orphaned pages to compare the security analysis against such that we can assess whether orphan pages are more, or less, vulnerable to common exploits. Once we make this control group, we perform the same security evaluation against pages of both groups. Concretely, we perform black-box security scans which make use of various fuzzing techniques to asses the susceptibility of a web page to security exploits. We then dive deeper into two of the most common vulnerabilities: Cross-Site Scripting (XSS) and SQL Injections (SQLi), while also shedding light on more minor issues such as bad practices in security configurations. The results of this evaluation allow us to answer **(RQ2)**.

After answering both research questions, we obtain the answer to the main question **(RQ)** of this thesis. The following section will summarize our findings of answering the research questions in the form of the scientific contribution made by this work.

**Answering the Research Question**

Throughout this thesis we will come back to these research questions. Whenever applicable, we will highlight in a chapter the contribution it makes toward the research questions of this study in the form of a special colored box like this one. Apart from answering the questions, this will also help the reader to put each chapter into perspective, and be reminded to the bigger picture of this research, allowing to better understand how each different part contributes to the whole. At the end of this thesis, we summarize these highlights and answer each research question more formally.

## 1.4. Summary of Contributions

In summary, we make the following contributions:

- We provide the first methodology for detecting orphaned web pages on a *single* domain, using only publicly available information.
- We perform the first large-scale detection of orphaned web pages in the wild, and report a lower bound on the prevalence of them. On a sample of 100,000 websites, we observe that at least 1,953 pages, spread over 907 domains, are orphaned, with some of the pages being as old as 20 years.
- We compare the security posture of orphaned pages to a control group of non-orphaned pages, and we find that orphaned web pages are more prone to vulnerabilities. For example, we find 7.1% of the orphaned pages are vulnerable to Cross-Site Scripting (XSS) attacks, which differs substantially to 0.9% of the control group being vulnerable ($p < 0.01$ using $\chi^2$).
- We share an implementation of our methodology that can be used for further research on this topic, as well as by website administrators and security professionals to audit websites. We contribute our implementation as open source.

## 1.5. Outline of the Thesis

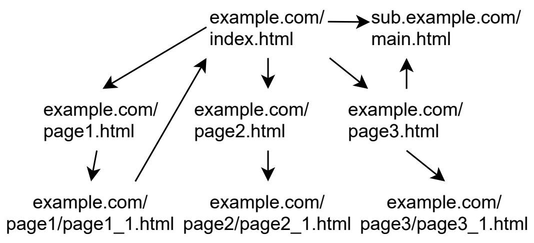
The remainder of his thesis is organized as follows: First, we provide the necessary background on orphaned web pages in Chapter 2. We explain the concept in more detail and provide some nomenclature to study the topic. The goal of that chapter is to answer **(RQ1.1)**. We then describe our methodology in Chapter 3 and elaborate on our evaluation and measurement study in Chapter 4. At the end of that chapter, we can answer **(RQ1.2)**. In Chapter 5, we present our results. We provide a lower bound on the prevalence of orphan pages to answer **(RQ1.3)**, we dive deeper into some types of orphan pages to answer **(RQ2.1)**, and we perform a security analysis to answer both **(RQ2.2)** and **(RQ2.3)**. We discuss our findings in Chapter 6, which allows us to answer the final subquestion, namely **(RQ2.4)**. We end with an overview of the related work in Chapter 7, and we conclude in Chapter 8.

# 2 Background

The general concept of orphaned resources has been studied before (see Chapter 7) and we briefly explained the concept of orphaned web pages in the introductory chapter. However, to the best of our knowledge, we are the first to comprehensively study orphaned resources within the context of web pages. As such, we lack an established nomenclature to study and discuss the topic. This chapter provides such nomenclature, along with a deeper background on the concept of orphaned web pages. We explain how we expect these pages to generally manifest, as well as discuss the potential security impact these pages can have.

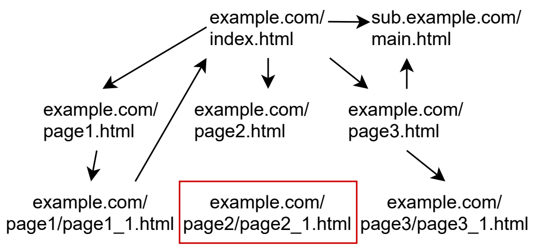
## 2.1. Definition of an Orphaned Web Page

We define an orphaned web page based on the sitemap of a website. Figure 2.1 shows a simplified example of such a sitemap graph: a starting node, called root or index (`example.com/index.html`), with edges (links) to children nodes (pages at `example.com/page1.html`, `example.com/page2.html`, etc.) and further descendants. A page becomes orphaned when all links (edges) to a page (node) are removed, and no other page (node) links to that page (see Figure 2.1b, where the page at `example.com/page2/page2_1.html` has no inbound link). However, although there is no path from the root, the page remains accessible via its URL. That is, a page is orphaned if it cannot be reached through graph traversal from the website's entry points –with a typical entry point being the website's index page– yet is still available by directly accessing the URL. The term 'orphan' forms here the parallel of a child node being removed as a descendant, which results in the child node further existing without any parent node.



example.com/ index.html
sub.example.com/ main.html
example.com/ page1.html
example.com/ page2.html
example.com/ page3.html
example.com/ page1/page1_1.html
example.com/ page2/page2_1.html
example.com/ page3/page3_1.html

(a) Site referenced



example.com/ index.html
sub.example.com/ main.html
example.com/ page1.html
example.com/ page2.html
example.com/ page3.html
example.com/ page1/page1_1.html
example.com/ page2/page2_1.html
example.com/ page3/page3_1.html

(b) Link removed

Figure 2.1: Sitemap example for a website at `example.com`. When `example.com/page2.html` stops linking to `example.com/page2/page2_1.html`, then the latter becomes orphaned.

## 2.2. Creating an Orphaned Web Page

Prior work suggests that an orphaned web page can be caused by misconfigurations (see Chapter 7). We illustrate three examples of how misconfigurations in website (re)structuring can lead to orphaned web pages.

### 2.2.1. Link Removal

The first scenario we touched upon when defining an orphan page, and explains how, by removing a single link, one page can become inaccessible by following links from the index page, yet remains available through its URL (see Figure 2.1b). This event occurs when the website administrator intends to fully remove the child page from the website, which requires both the removal of the page, as well as the removal of all links on the domain pointing to that page to avoid errors and confusion. If only the latter step is executed without the former, we end up with an orphan page.

This is problematic: since the website administrator carries on with the false belief that the child page cannot be accessed. The orphaned web page might then not be maintained anymore and becomes outdated, making it more prone to vulnerabilities over time.

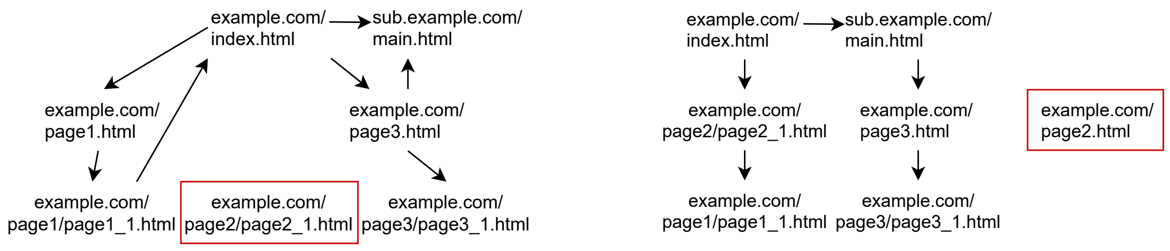
A less harmful scenario, yet similar, is when a link gets accidentally removed without the intention of removing the page it points to. In this case, mere confusion might be caused due to the missing link, but the page itself will likely be maintained along the same standards as other pages of the domain.

### 2.2.2. Page Removal

Similarly as with link removal, if we would remove the page that contains the link, the link itself gets removed as well, resulting in an orphaned child page (see Figure 2.2a). As with the previous scenario, we can differentiate here between the case when the child page is meant to be removed as well, and when it is not. In the former, we create an orphaned web page, and strand again in the situation where a website administrator is unaware of one of its pages still being online, leaving it therefore potentially unmaintained and vulnerable. In the latter scenario, the website administrator can provide a direct link from the parent of the removed web page to the child of the removed web page. Note, if this is not done the child still becomes orphaned, however again with the likelihood that the child page will be maintained along the same standards as other pages of the domain.

### 2.2.3. Website Restructuring

Occasionally, website administrators may want to restructure their website. Depending on the website's size and construction, this can be rather intricate. During this process, the administrator might neglect the re-linking of a page, which can cause this page to not be reachable through links from the index page. The page then becomes orphaned (see Figure 2.2b). If, however, this was not intended, the page can again become unmaintained over time, causing a potential threat to the security of the website.



(a) Page removed

(b) Website restructured

Figure 2.2: Other scenarios in which a page can become orphaned.

## 2.3. Types of Orphaned Web Pages

In the previously described scenarios we notice a distinction between two categories of orphaned pages: unmaintained orphaned pages and maintained orphaned pages. They form a key difference in this thesis, as we are focused on detecting and analyzing unmaintained orphaned pages due to their presumed susceptibility to vulnerabilities in outdated software.

### 2.3.1. Unmaintained Orphaned Web Pages

Unmaintained orphaned web pages are those where the administrators are no longer aware of their existence (or do not care), and, consequently, do not apply (security) updates to the underlying application(s) providing the page. For example, this can happen if a team runs a website for a limited time (e.g., a product promotion or recruitment campaign, as the Deloitte example in Chapter 1 illustrated) or if the person responsible for maintaining a website leaves the company. With the lack of updates, the orphaned web page might become outdated, possibly making it prone to vulnerabilities over time.

Coming back to the quadrants of Figure 1.1, we can classify unmaintained orphan pages on the right side of the matrix (quadrants 1 and 4). In quadrant 1, a user still knows the URL of the orphaned page, and can access it directly. This means that these web pages are detectable as their URL is known by *someone* or recorded *somewhere*, like, for example, in an old messaging board post or in archival data.

Pages in quadrant 4 are much more challenging to detect. These pages are orphaned and unmaintained, but they were never intended to be used by the public. This means that their specific URL may never have been referenced on a publicly accessible site, or that the page is hidden behind (simple) authentication. In the former case, the page could theoretically be accessed if one could guess the URL or brute-force it, but this is no small feat and akin to searching for the needle in the haystack, when there might not even be a needle in the first place. Correspondingly, we focus on orphaned pages previously known to the public (quadrant 1), and we consider pages unknown to the public (quadrant 4) out of scope.

### 2.3.2. Maintained Orphaned Web Pages

Web pages may also be orphaned without becoming unmaintained. This can be the case if a link is unintentionally removed. In this case, the orphaned page is unreachable if one only follows links from an entry point on the website. However, the administrator is still aware of the pages and maintains them along the same care as other pages. This is also the case if the orphaned page utilizes the same framework or content management system as the main site.

Maintained orphaned pages correspond to the left quadrants of Figure 1.1 (quadrants 2 and 3). These pages are generally less of an issue in terms of security vulnerabilities, although they can be inconvenient to users or cause confusion.

## 2.4. Security by Obscurity

In addition to accidentally orphaned pages, whether maintained or not, a page might become orphaned by design, with the intention of "protecting" the page against various attack vectors. Operators may chose to "hide" a page by removing links to it, excluding it from public search (using `robots.txt`) to reduce the attack surface, and only sharing it's URL with a select group of (trusted) people. This may happen for publicly available internal applications, or for pages for which a vulnerability was discovered. Although this might result in less exposure and less traffic to the vulnerable page, security by obscurity is known to be largely ineffective, and the site actually remains vulnerable to exploitation by adversaries. While one could consider these sites as "maintained" orphaned pages, we argue that if they are hidden with the goal of security by obscurity, and administrators are not actually applying other security updates, then they provide additional attack surface (unlike other maintained orphaned pages), and they are actually unmaintained, or at the very least insufficiently maintained.

## 2.5. Security Impact of Orphaned Pages

We expect that unmaintained orphaned web pages are more prone to vulnerabilities, due to delayed updates (or not receiving them at all) and pre-dating modern defenses. While the main infrastructure of the website is likely maintained, such as the underlying operating system or webserver (e.g., nginx or Apache), we see no reason to assume that administrators are patching vulnerabilities of orphaned web pages hosted on the infrastructure. This also holds true for server-side applications only used by the orphaned pages, as the administrator expects that the functionality is not reachable anyways, and, thus, they have no incentive to expend time and resources to apply security patches. Naturally, this differs for systems applying automatic updating, though these automatic updates may break over time or are only partially applied (e.g., configuration files are part of software packages on Linux distributions, but they are rarely updated by automatic updates, and misconfigurations and insecure configurations remain even with automatic updates enabled).

Moreover, a website administrator can easily build up a false sense of security. If a technology stack is changed because it used to be vulnerable, the administrator might consider the website safe from specific classes of vulnerabilities. For example, a website might migrate from pages generated dynamically on the server-side to statically-generated pages, with the dynamically-generated pages becoming unmaintained orphaned. The administrator may then think the website is secure from server-side code injection attacks or path traversal vulnerabilities because no code is being executed on the server-side anymore. However, as these pages have only been orphaned and not truly removed, this assumption would be a mistake. Consequently, an unmaintained orphaned web page can become the Achilles' heel of the entire website.

**Answering the Research Question**

(**RQ1.1**): *How do we define and characterize orphaned web pages?*
We define the concept of an orphaned web pages as: a page that is no longer referenced to by any other page on the website, while still remaining directly accessible through its URL. A key characteristic of an orphaned web page is, hence, that at one point in time the URL of the page was somewhere (publicly) available online. We leverage this concept in the next chapter to answer (**RQ1.2**) when we design our methodology for detecting orphaned web pages in the wild.

# 3 Methodology

With the background of Chapter 2, we are now equipped with the necessary onomastics to describe and study the topic of orphaned web pages. Furthermore, we have gotten insight into the nature and characteristics of orphaned web pages, therefore answering **(RQ1.1)**. We now build upon this knowledge, and proceed with designing a methodology to detect these pages in the wild, and at-scale.

## 3.1. Orphaned Page Identification Methodology

We define orphaned web pages as pages with no ancestry links in the domain's sitemap (see Chapter 2). That is, when embarking from the root and following links, no path exists to reach the orphaned page. Consequently, this means we cannot rely on any information on the domain/website to identify orphaned pages. One approach to solve this is to simply 'guess' pages that might be orphaned by brute-forcing URLs. This would entail probing the domain for all permutations of valid URL characters up until a certain URL-length, in the hopes of finding active pages that are no longer advertised on the website. Besides being a cumbersome method, this approach yields no guarantees for finding truly orphaned pages –if it would find any pages at all– and is impossible to scale for Internet-wide measurement studies. We therefore need a more tailored approach for finding candidate URLs of potential orphan pages that we can later probe to test for their orphan status.

We can achieve this tailored approach by making use of archived data from the web. Web archives are large databases that contain previous versions of websites. These versions are obtained by systematically crawling (part of) the World Wide Web and archiving the results (more details on web archives can be found in Section 4.1). Leveraging this archived data, we learn how a domain used to look like throughout the years, not only in terms of content, but also in terms of its structure, that is, its sitemap. This allows us to compare previous versions of a website to its current version, and see whether any of the paths in its sitemap have been removed. For a historic perspective on which sites used to be reachable via the root, we thus leverage the Internet Archive (IA) [3], which provides access to a time-stamped history of websites.

Correspondingly, we can identify potential candidate orphan pages by investigating pages that were once part of a domain's sitemap, but are not part of it anymore. We can then probe these candidate pages to determine if they are still accessible, and whether their content is different from the last archived version. Using this methodology, we detect pages from quadrant 1 of Figure 1.1.

Generally, a website administrator might advertise (part of) a website's structure in an XML file called `sitemap.xml`. This provides a list of pages on the domain that a search engine can crawl and index. This is ideal for our study, as we can directly compare the historic and current version of a domain's sitemap, and identify the discarded paths in the sitemap leading to orphaned web pages. Unfortunately, not every website makes their `sitemap.xml` available. Some that do might have them excluded from the Internet Archive (see Section 4.1). Therefore, we cannot rely on a pre-constructed sitemap, but instead need to reconstruct it by retrieving the full list of archived web pages from the Internet Archive, and using it to determine which URLs might contain potentially orphaned web pages.

### 3.1.1. Gathering Candidate Orphan Pages

First, we utilize the Wayback CDX Server API [30] to retrieve the archived data. It allows us to query a domain, after which the CDX Server returns the list of archived pages for that domain, including subdomains. From this list, we collect two sets: a current set, and a past set. The current set contains all the pages encountered by the crawler in 2020, the past set all pages last encountered before 2020. Because the Internet Archive makes use of crawls (meaning, it traversed the website from the root or inspects the domain's `sitemap.xml`), we know that each page is reached via a path starting from the root or it is listed in the sitemap. If a page was listed before, but is not present in the 2020 crawls, it has either been removed, was ignored by a later crawl, or got orphaned because the ancestry link got removed in some sort of way (see Chapter 2 for the different ways in which a page can become orphaned). To distinguish among the options, we identify orphaned pages by liveness probing after pre-filtering candidates locally, which will later be discussed in detail in Section 3.1.5.

We do have to account for the fact that the Internet Archive also stores pages that caused redirects, server errors, or client errors during the initial crawl. Considering that we are only interested in reachable pages, we query the API to exclusively return the list of web pages that responded with a HTTP status code OK (200) in their original crawl.

### 3.1.2. Discarding Resource Files

We focus our analysis of orphaned pages on actual web pages. However, domains may host additional resources, such as documents, pictures, JavaScript code, stylesheets, etc., which are of no interest to us. Thus, we filter the list of candidate orphan pages by removing URLs ending in a known resource-related file extension (see Appendix A for the full list).

### 3.1.3. Dynamic URL Detection (DUDe)

Dynamically generated URLs are a common challenge when identifying any kind of resource on the Internet. This can occur when news sites make articles available under unique URLs, but they are only reachable from the index page while they are recent. For example, for archived news articles, often some form of (chronological) structure is used for their URLs, like `https://example.com/articles/1997/January/name-of-the-article.html`. This results in many URLs containing a similar prefix with a (possibly generated) suffix. If we would probe all these URLs, we would put unnecessary load on webservices. Moreover, it would significantly extend the runtime of our methodology, while these URLs are not actually as interesting to us as they are likely not truly unmaintained orphaned, but are likely actually maintained.

Therefore, we use a heuristic to identify the common prefixes of these URLs and remove them from our list before probing for liveness. If a page contains many long links, we try to identify a common prefix based on character frequency. We do so by counting the frequency of each character at an index of the URLs, and generating a prefix based on the most frequent character for each position (in case of a tie, the first one encountered will be used). We then shorten the prefix, one character at a time, until it is general enough (found in at least 5% of links, see below), or until we consider it too short to be a valid prefix. We repeat this process until we can detect no more prefixes. Naturally, this approach limits the number of dynamic orphan pages we can observe with our approach. Nevertheless, following our previous observation on not putting unnecessary strain on networks and systems, we consider it crucial to exclude dynamic URLs.

Before walking over an example of DUDe, we first describe our heuristic in pseudocode, which can be found in Algorithm 1. It has four parameters:

1. **Popularity cutoff (PC)**: the percentage of URLs on a domain that need to contain the prefix.
2. **Short-link threshold (ST)**: the amount of characters a link needs to have to be considered short.
3. **Long-link threshold (LT)**: the amount of characters a link needs to have to be considered long.
4. **Long-link cutoff (LC)**: the amount of links on a domain that need to be long for the heuristic to run.

We determine these parameters from a random sample of 1,000 domains, taken from our input data (see Section 4.1). We evaluate the following values: [5%, 10%, 15%, 20%, 25%, 30%] for the popularity cutoff, [5, 10, 15, 20] for the short-link threshold, [20, 25, 30, 35, 40] for the long-link threshold, and [0, 3, 5] for the long-link cutoff. We optimize for the highest percentage of reduction. After evaluating all permutations, we obtain 5% for the popularity cutoff, 15 for the short-link threshold, 20 for the long-link threshold, and 0 for the long-link cutoff, as the optimal input parameters. This configuration reduces on average the number of URLs per domain by 67%, with a standard deviation of ±30 percentage points and median of 73%.

**Algorithm 1** Dynamic URL Detection

```
1: $max\_len \leftarrow 0$
2: $avg\_len \leftarrow 0$
3: $large\_links \leftarrow \{\}$
4:
5: for $link$ in $page$ do
6:     if $len(link) > (LT + len(dom\_name) + 8)$ then    ▷ Check the length, 8 represents len(“https://")
7:         $large\_links.append(link)$
8:     Find average and max length of all links in sitemap, and store in $avg\_len$ and $max\_len$
9: if $len(large\_links) \leq LC$ then return
10: for $link$ in $large\_links$ do
11:     for $c$ in $link$ do
12:         Count character frequency at each position
13:
14: $generated\_link \leftarrow$ “"
15: for $i = 0..max\_len$ do
16:     Append most occurring character at position $i$ to $generated\_link$
17: $prefix \leftarrow generated\_link[: avg\_len]$    ▷ Cut the prefix to $avg\_len$ characters
18:
19: $blocklist \leftarrow \{\}$
20: $allowlist \leftarrow \{\}$
21: do
22:     for $link$ in $page$ do
23:         if $prefix$ in $link$ then
24:             $blocklist.append(link)$
25:         else
26:             $allowlist.append(link)$
27:     $prefix \leftarrow prefix[: -1]$
28: while $len(blocklist) < PC \cdot len(sitemap)$
29:
30: if $len(prefix) < (len(domain\_name) + 8 + ST)$ then    ▷ Check the length, 8 represents len(“https://")
31:     Restart procedure, ignoring links containing $prefix$
32: else
33:     Write out $allowlist$ and $blocklist$
34:     Start procedure again on $allowlist$
```

```
https://example.com/articles/1997/January/article1.html
https://example.com/articles/1997/January/article2.html
https://example.com/articles/1997/February/article1.html
https://example.com/articles/2002/May/article1.html
https://example.com/articles/2002/May/article2.html
https://example.com/articles/2002/May/article3.html
https://example.com/path/on/domain.html
https://example.com/contact.html
https://example.com/login.html
```

Listing 3.1: URL list before Dynamic URL Detection.

```
https://example.com/path/on/domain.html
https://example.com/contact.html
https://example.com/login.html
```

Listing 3.2: URL list after Dynamic URL Detection.

Optimizing for the highest reduction percentages may filter out unintentionally orphaned web pages by error. We consciously choose this optimization to provide a lower bound on the prevalence of unintentionally orphaned pages in the wild.

**DUDe Example:** We illustrate how our algorithm removes dynamic URLs for an example domain, see the set of URLs in Listing 3.1. Here, we first check for long links. We continue by counting the character frequency at each position, followed by generating a URL with at each position the most encountered character of that position. In our example, this becomes "`https://example.com/articles/1997/May/articlei.htmlhtmll`." We then shorten the URL to the average URL size among the URLs on the domain, which we use as a prefix. This prefix URL is "`https://example.com/articles/1997/May/articlei`." Since no URL matches this prefix, we shorten it one character at a time until at least 5% match the prefix, or until the prefix is too short. In our example 5% corresponds to 0.45 pages, which is rounded up to 1 page. In practice, this will be a (much) higher threshold as websites typically have more than nine pages.

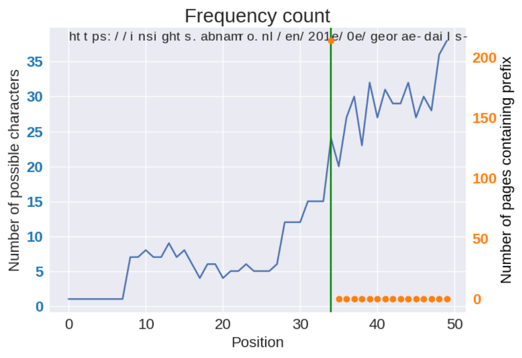
We shorten the prefix until it becomes "`https://example.com/articles/1997/`," which is present in three URLs. We then remove the URLs containing the prefix and repeat the process on the remaining URLs. The second generated prefix is "`https://example.com/articles/2002/May/arti`," which does not require further shortening as it is matched by three of our URLs. These URLs are removed, after which the algorithm stops since no further long links are present in our set, resulting in three URLs remaining (see Listing 3.2).

### 3.1.4. Visualization of DUDe

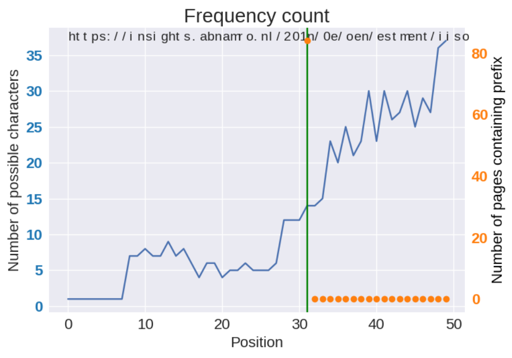
Walking through an example of DUDe on an actual domain with more than 9 pages is cumbersome and long-winded. To therefore illustrate the performance of our heuristic in the wild, we visually represent the process in Figure 3.1. We take as an example `abnamro.nl`. Each plot in Figure 3.1 represents the process for each generated link. The left axis depicts the number of possible characters for each position in the URL based on the frequency count of all links in our data set for `abnamro.nl`. Naturally, we see a rather low number of possibilities for the first part of a URL (which all have a prefix starting with `https://` followed by a subdomain of `abnamro.nl`). Toward the end of the URL we see the graph increasing, signifying the increase in entropy for character possibilities in the suffix.

The vertical line in the plot shows the prefix-cutoff at which point our heuristic either found enough matches for the prefix (green line) or the prefix has become too short (red line). This choice is further depicted by the orange dots counted on the right-hand axis. Concretely, each dot illustrates a prefix-cutoff for the position in the generated URL (signified by the x-coordinate) along with the number of URLs in the data set that match this prefix (signified by the y-coordinate). Indeed, the shorter we cutoff the prefix (i.e. the more we go to the left in the graph), the more matching URLs we obtain in the data set.

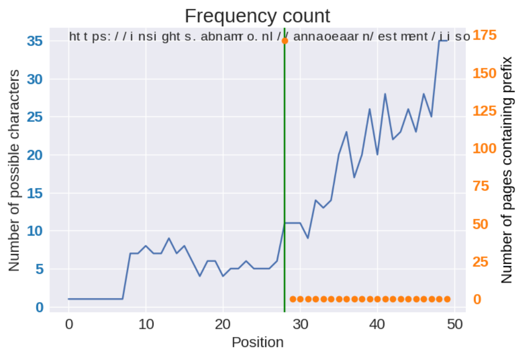
For the example in Figure 3.1 we found three prefixes that could be used to filter out dynamically generated links.



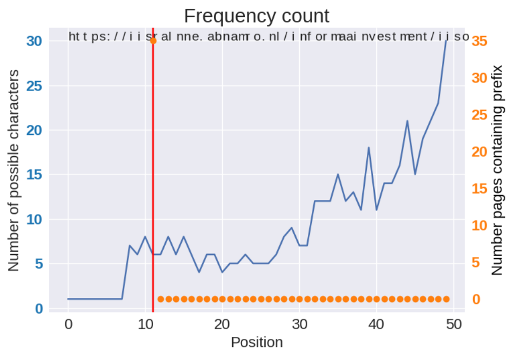
(a) First Iteration



(b) Second Iteration



(c) Third Iteration



(d) Fourth Iteration

Figure 3.1: Each plot shows the result of running Algorithm 1 on a list of pages for `abnamro.nl`. To put the graphs into context, the generated link is printed on top, with each character aligned to the proper position on the x-axis. The blue line depicts the amount of different possible characters at each position of a link. The vertical line shows the cutoff made by the algorithm for where the prefix stops (i.e. everything on the left of the line is considered the prefix). If the line is green, the prefix is used to filter. If the line is red, the prefix is not used and the algorithm is stopped. Finally, the orange dots show the amount of pages that contain the generated link, for each cutoff until the actually used prefix.

### 3.1.5. Probing Liveliness of Candidate Orphan Pages

After filtering the list of potential orphan pages with DUDe, we analyze the pages by probing them. This allows us to exclude pages that were actually taken down after their links have been removed. We perform a HTTP HEAD request to retrieve the HTTP status code of the web page. We discard pages producing error responses (4xx and 5xx status codes), redirects (3xx status codes), as well as any page that does not return the HTTP status code OK (200).

**Answering the Research Question**

(**RQ1.2**): *How to systematically detect orphaned web pages in the wild and at Internet-scale?*
With our open source implementation, we have designed and developed a methodology (based on the answers found to **(RQ1.1)**) which is able to detect orphaned web pages in the wild, and at Internet-scale, therefore answering **(RQ1.2)**. We leverage web archive data to identify web page that were once referenced by their domain, but are no longer captured by recent web crawls. By probing these pages for liveliness, we obtain a list of potential orphan pages. Using various filter techniques and heuristics, we make our methodology scalable and fit for Internet measurements. Next, we can proceed to evaluate our methodology by performing a large-scale measurement study for orphan pages in the wild. Correspondingly, this will allow us to find a lower bound on the prevalence of these pages, therefore answering **(RQ1.3)**.

### 3.1.6. Open Source Implementation

We provide a dockerized open source implementation[1] of our methodology. It implements our methodology for running it against a single website (see Figure 3.2), making large-scale studies embarrassingly parallel (using, for example, GNU Parallel [55]) and enabling them to scale horizontally across machines easily.

The open source nature of our implementation also allows security professionals and researchers to tailor our methodology to their needs. For our large-scale measurement study, we chose cutoff and threshold parameters that led to a high reduction rate to obtain a lower bound. However, when used on a single domain, security professionals may prefer a more lenient Dynamic URL Detection, or even omit this module from the procedure entirely, to apply additional context-aware filters on the list of potential orphans. We expand on use cases of our technique later in Chapter 6.

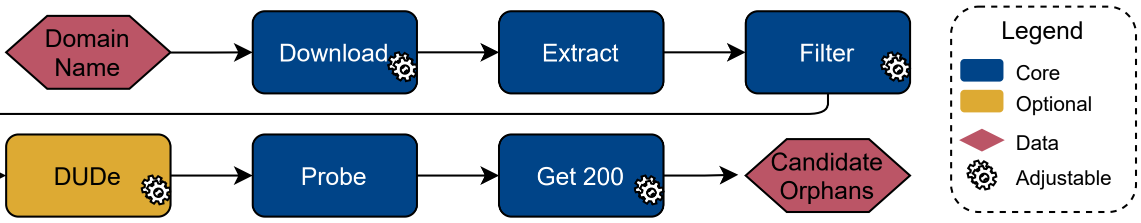


Figure 3.2: Overview of our implementation.

[1]Available at `https://github.com/OrphanDetection/orphan-detection`

# 4 Evaluation and Measurement Study

In this chapter, we describe the large-scale measurement study done for this research. It evaluates the scalability of our method designed in the previous chapter. We begin this chapter with a depiction of the data sets used for this study. Furthermore, we report on the timeline and implementation of our measurements as well as the solutions we implemented to tackle issues encountered during the measurements. We deploy our methodology on 20 servers within our university network, and split our study into two phases: downloading and processing.

In the downloading phase, we first gather all archive data for each domain in our data set. Once this is finished for each domain, we then proceed extracting the candidate orphan pages, applying the filter and heuristic, and probing for liveliness. Although our methodology was described for use on a per-domain basis, we perform this split to ensure that the downloaded data is consistent in date and not scattered over time due to the need for processing each domain (i.e., filtering and probing) before downloading the next one. This is crucial for our measurement study since we analyze the collective results and not just the per-domain conclusions. A summary of the timeline and reduction results of our measurement study can be found in Table 4.1.

Table 4.1: Summary of our large-scale measurement.

| **Step** | **Timeframe** | **Result** |
|---|---|---:|
| Downloading archive data | December 16, 2020 - December 20, 2020 | 6,092,214,431 Pages |
| Filtering file extensions | December 20, 2020 | 4,033,539,860 Pages |
| Dynamic URL Detection | December 29, 2020 | 924,190,351 Pages |
| Probing and extracting pages with HTTP 200 response | January 1, 2021 – January 29, 2021 | 36,442,679 Pages |
| Size-based filtering | February 14, 2021 | 1,821,682 Pages |
| Removing Invalid Pages | February 23, 2021 | 1,257,063 Pages |

## 4.1. Large-Scale Dataset

We rely on two different data sets for our measurements: first, a random sample of 100,000 domains from the Tranco Domain List [33], and, second, the archived data for these domains from the Internet Archive. Before discussing both, we first elaborate on the reliability of Top 1M list when used for scientific research.

### 4.1.1. Reliability of Top 1M Domain Lists

Though many studies within the security community have relied on popular 1M lists –such as Alexa Top 1M, Cisco Umbrella Top 1M, and the Majestic Top 1M–, research has shown that traditional Top 1M lists are less reliable than initially thought [46, 33, 45]. Concretely, most Top 1M lists are not stable, meaning that their ranking can differ significantly among two consecutive days or weeks. They have been shown to contain unresponsive websites, and are often vulnerable to manipulations, making it possible to promote one's domain within the Top 1M list, therefore falsifying the ranking. Additionally, it was demonstrated how most of these sets actually disagree on the exact list and order of domains, with one study showing a meager 2.4% overlap among the agreed order of the four Top 1M lists [33].

### 4.1.2. Tranco Domain List

To counter the existing issues with Top 1M sets, we use the Tranco list [33]. It averages the ranks of domains, as listed by Alexa, Cisco Umbrella, Majestic, and Quantcast. The domains are averaged over a period of 30 days, and ranked according to the Dowdall rule. Although the Tranco data set has a particular focus on battling the manipulation of Top 1M lists, it also ensures solutions for other shortcomings, such as better agreement on the popularity of domains, and stability. This makes the list more resilient to manipulations and less prone to fluctuations.

For this study, we use the Tranco list from 14 December 2020.[1] We evaluate our methodology on a subset of the Tranco list, consisting of a random sample of 100,000 domains taken from the top 500,000 ranked domains. The exact list of domains is available at our open source repository.[2]

### 4.1.3. Internet Archive

To detect orphan pages, we make use of the Internet Archive, which is an online digital library, archiving and providing access to various resources including websites [3]. We query the Internet Archive's Wayback Machine using the CDX API [30]. It stores a URL key, timestamp, full URL, media type (MIME), status code received during crawl, a message digest, and the length. However, we only retrieve the timestamp and full URL, and do this only for pages that returned a status code of OK (200) upon their initial crawl; this makes our approach lightweight in terms of archive data.

## 4.2. Downloading Phase

For the downloading phase, we take our input domains and retrieve the archive file for each domain. We retrieved the archives between December 16, 2020 and December 20, 2020, with ten downloads in parallel on each of our servers, with a delay of one minute between each batch. We do so to reduce the load on their servers in consultation with the Internet Archive.

When fetching the archive data for a domain, we encounter four different issues: (1) the server throws an error, (2) the server is temporarily offline, (3) the server sends back 0 bytes, and (4) we hit the rate limit. Whenever we encounter an issue, we move the corresponding domain to the back of our queue and attempt to retrieve it again later. We repeat this process until we retrieved all archival data successfully, or we are left with cases of only returning 0 bytes or an error (for which we then assume the data is not available in the Internet Archive).

Overall, we retrieve sitemaps for 96,537 domains from our sample of 100,000 domains (with the remaining domains not being indexed). The total comprises a storage size of around 1.2 TB, which is good for a total of 6,092,214,431 URLs, yielding on average 60,922 pages per domain (median: 6,955, standard deviation: 126,094).

## 4.3. Processing Phase

We then apply our methodology on the pages from the 96,537 domain archives. Table 4.1 shows an overview of timeframe and results of our large-scale measurement, which we discuss next. Filtering the pages on our list of file extensions, we first obtain a total of 4,033,539,860 potential orphan pages. After removing dynamic URLS with DUDe, we reduce the list to 924,190,351 URLs (a reduction of 77%).

### 4.3.1. Probing

We probe the remaining URLs to determine which pages still return a status code of OK (200). To avoid overloading the domains with our probes, we shuffle the list of URLs and split them over our deployment. This ensures that, on average, we do not exceed 87 requests per domain per hour. Moreover, our probes used a customized User-Agent (Listing 4.1) that allows administrators to easily opt-out from our study at any time (see also the ethics discussion in Section 6.4).[3] We probed candidate orphaned pages from January 1, 2021 to

[1]Available at `https://tranco-list.eu/list/5Q3N`.

[2]Available at `https://github.com/OrphanDetection/orphan-detection`.

[3]During probing, we received two requests from administrators to remove their website from our study. Correspondingly, we excluded some links from probing.

```
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:80.0) Gecko/20100101 Firefox/80.0
    ↝ TUDelftNetworkMeasurement/2020 (contact; t.fiebig; at; tudelft.nl; s.r.g.pletinckx; at;
    ↝ student.tudelft.nl)
```

Listing 4.1: User-Agent used in measurement probes.

January 29, 2021. After probing and removing pages that did not respond with a HTTP status code OK (200), we are left with 36,442,679 candidate pages (a reduction of 96%).

Figure 4.1 depicts the distribution of the responses received during our probing. As expected, most of the pages are in fact removed from their domain, along with their respective links pointing to the page. However, there is still a significant set of pages that yield a valid response, which might indicate that some service still actively resides behind the URL. Many pages respond with a 2xx, of which a majority give the common OK (200) status code (note the log scale of the graph). Although we only focus on this particular response code, we do like to note that the other status codes can potentially indicate the presence of an orphan page.

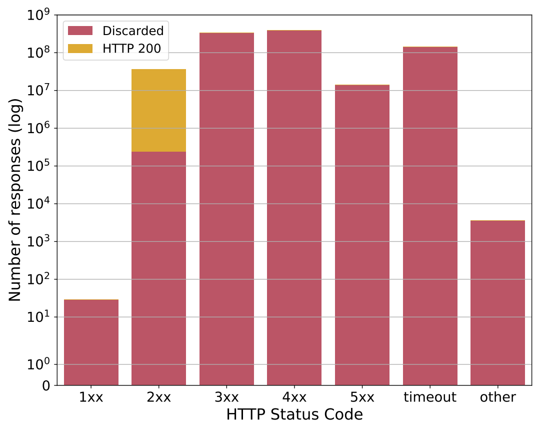


Figure 4.1: HTTP status codes encountered during the probing the potential orphaned pages. The chunk of pages that responded with a status code of 200 is highlighted yellow.

The group of pages responding with an OK (200) status code provide the highest guarantee of having an active page behind the URL. Moreover, they require no further parsing/processing of the response to eventually reach an active page (something that is necessary with, for example, a 3xx redirect response). This also keeps our method relatively lightweight, and avoids unnecessary overhead. Parsing the other responses, however, might nonetheless lead to an orphaned web page, which we want to illustrate for completeness in the following example.

Let's take a page that used to be advertised on a website, and that initially responded with a status code OK (200). The Internet Archive then crawls the page, and stores its content along with the recorded OK (200) response. Shortly after, the website administrator makes the page redirect to an updated version of the page, residing behind a different URL. Before the Internet Archive can perform a new crawl, the website administrator restructures the website, rendering both pages as orphaned (see Chapter 2 for ways in which a page can become orphaned). The updated page never had an advertised URL on the domain, and hence does not have an entry in the Internet Archive. The initial page, however, is registered in the Internet Archive, be it with a status code OK (200). Our probing would scan this URL, but register it as a 3xx redirect since the website administrator changed its response behavior before the next archive-crawl. Following through on this redirect will, however, bring us to an unmaintained orphan page since both pages were removed during the restructuring of the website. This shows that other responses might be worth exploring, be it with additional overhead to our methodology.

Given the modularity of our methodology, it is easy for users of our implementation to incorporate more status codes, resulting from the probes, in further analysis steps. We elaborate more on this in Chapter 6.

Figure 4.2 shows the distribution of OK (200) responses per domain. From the 96,537 probed domains, 21,654 domains (22.4%) had a page that returned a status code of OK (200). Indeed, as we can observe from the figure, the majority of our domains have between 1-70 pages that are no longer listed on the Internet Archive, yet return a status code of OK (200). The standard deviation is rather big (12,332.23), leading to

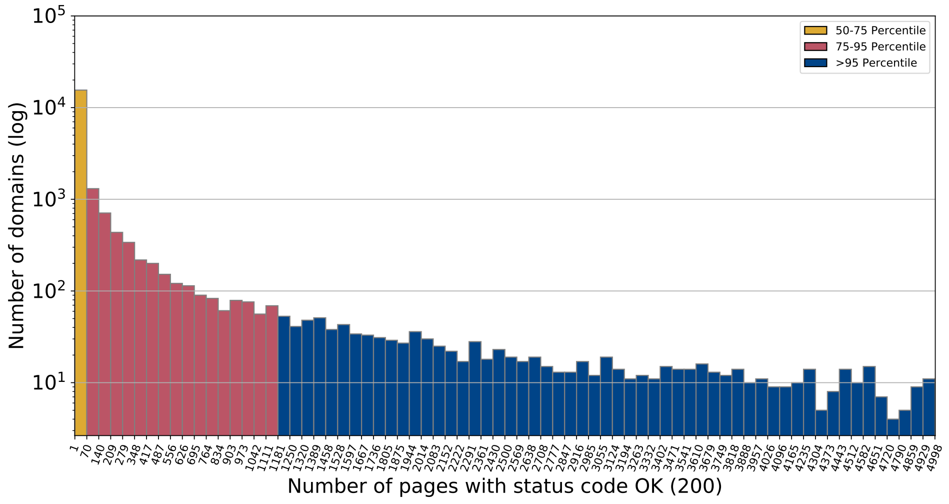


Figure 4.2: Distribution of the pages per domain returning a status code of OK (200).

a mean value of 1,682.95 pages per domain. However, we obtain a median of 11, which is a much more representative value.

### 4.3.2. Identifying Custom Error Pages

Unfortunately, not every unavailable web page returns a status code of Not Found (404), or Redirect (301/302). Instead, they might return a custom error page stating the page was not found, while the HTTP status code is actually OK (200). In fact, while probing pages, we regularly encounter discarded pages (i.e., web pages displaying a standard HTML stating a similar message to “This page no longer exists”) returning a status code of OK (200). One way of removing such pages would be to perform a content analysis of the pages, and detecting messages such as “Page Not Found” based on a dictionary of strings. Unfortunately, this approach is not scalable for the magnitude of data we are still handling at this phase. As such, we take a different approach in removing these false positives at-scale

We retrieve the size of each page and remove pages from the same domain with a similar size (e.g., within 5 bytes of difference). The advantage of this method is that it requires only one traversal over the list of URLs, with each traversal doing basic operations that are fast in time. Algorithm 2 shows a solution that runs in $\mathcal{O}(nlog(n))$ time. As such, we can filter our full list of URLS efficiently and at-scale.

Leveraging size-based filtering, we remove pages that should have returned an error response, but responded with status code OK, and we reduce the set to 1,821,682 URLs (a reduction of 95%). Naturally, we may remove some pages with genuine content that is similar among multiple pages, such as login portals. However, in this work, we aim to determine a lower bound on the prevalence of orphaned pages, and, thus, we believe that omitting some pages and duplicates is an acceptable trade-off (see also Section 6.5).

**Algorithm 2** Similar page-size filtering

```
$to\_remove \leftarrow \emptyset$
$tmp\_remove \leftarrow \emptyset$
$current\_domain \leftarrow$ “"
$current\_size \leftarrow 0$
$\varepsilon \leftarrow 5$
for $index, domain, size$ of $urls.sorted(domain, size)$ do        ▷ Sort first on domain, then size
    if $domain == current\_domain$ then
        if $size == current\_size + \varepsilon$ then
            $tmp\_remove.append(index)$
            $current\_size = size$
            continue
    else
        $current\_domain = domain$
    if $len(tmp\_remove) \geq 2$ then
        $to\_remove = to\_remove + tmp\_remove$
    $tmp\_remove \leftarrow \emptyset$
    $current\_size = size$
    $tmp\_remove.append(index)$
```

### 4.3.3. Removing Invalid Pages

As a final check, we take a look at the filetypes of the pages in our data set. Previously, we did a rudimentary filter based on the extension in the url (such as .pdf, .xls, .pptx, etc.). Evidently, not all URLs will contain the file extension in their string. We therefore perform an additional check to ensure we only work with HTML-pages in our data set. This is done using the Linux `file -i [filename]` command in Bash. If the page is not of the type `text/html` we discard it from our data set. During this process, we also remove all pages containing a bad file encoding. In total, we discard 564,619 invalid pages leading to 1,257,063 pages (a reduction of 30%) that we need to analyze in more depth.

# 5 Analysis

In this chapter we further analyze the results from our large-scale measurement study. We are left with 1,257,063 pages for which we suspect they might be orphaned from their domain. The goal of our analysis is to find clear indicators that some of these pages have been unmaintained for several years.

First, we take a step back and analyze the nature of our downloaded archive data. This helps us in getting a better understanding on how these pages evolved over time, as well as what some of the characteristics are of the current sitemap. This is followed by a similarity check for all pages between their current version and their latest archived version. The goal of this check is to see how pages have changed over the years, and to spot the ones that appear to be unchanged since their last archival.

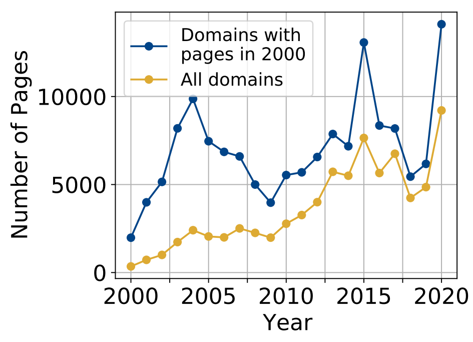
Following this analysis, we dive deeper into the content and structure of these pages. Concretely, we classify them based on characteristics of the page that indicate an unmaintained nature. The results of this analysis gives us an official lower bound on the prevalence of orphaned web pages, therefore answering **(RQ1.3)**.

We end this analysis chapter with an evaluation of the security of the found orphan pages. We test for basic vulnerabilities and compare it against a control group of non-orphaned web pages. We use this to answer most of the sub-questions to **(RQ2)**. The results of this evaluation are used in the discussion of Chapter 6 where we describe the impact orphan web pages can have on the security of web applications.

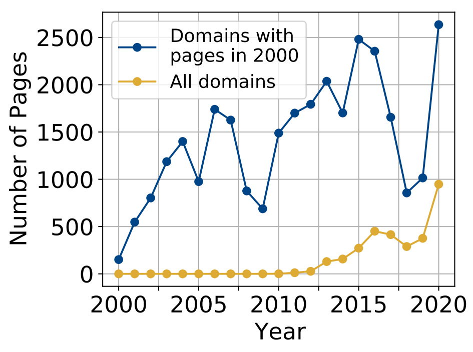
## 5.1. Archive Data Analysis

After collecting the archive data for the domains in our data set, we can investigate how domains evolve over time. Figure 5.1a shows the mean amount of pages per domain for each year between 2000 and 2020. Here, we see a clear increasing trend throughout the years, suggesting that websites grow. To account for bias inflicted by relatively young websites (since these would have zero pages before their first appearance online) we also show the growth for all domains that had at least one page archived in 2000. We observe that accounting for age still shows an increasing trend for the number of pages on a domain. This is also true for the median (see Figure 5.1b). Additionally, Figure 5.1c shows a boxplot for the amount of pages per domain per year. Apart from confirming the increase in the number of pages per domain over time, the latter two graphs also show the high variance in pages per domain. Especially from the boxplots, we can detect a significant difference between the first quartile and the third quartile.

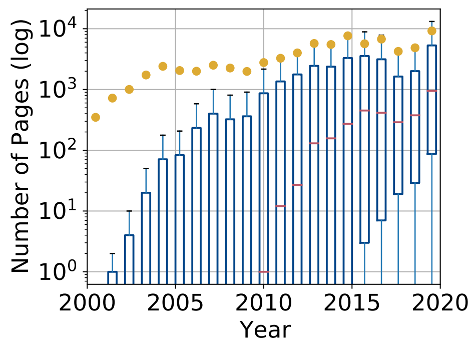
We described earlier that the 2020 sitemaps function as the basis for gathering the candidate orphan pages (see Section 3). We now take a look at how, and specifically when, the 2020 sitemaps came to be. Figure 5.1d depicts two Cumulative Distribution Functions (CDFs) of when a page of a 2020 sitemap was first seen in the archive data. The non-normalized CDF is calculated by summing the pages across all domains for each year and constructing a CDF of the sum. The normalized CDF is calculated by constructing a CDF for each domain, followed by taking the average percentage value across all domains for each year. Combining these years constructs the normalized CDF. Both are only calculated for pages that were last seen in the archive in 2020.



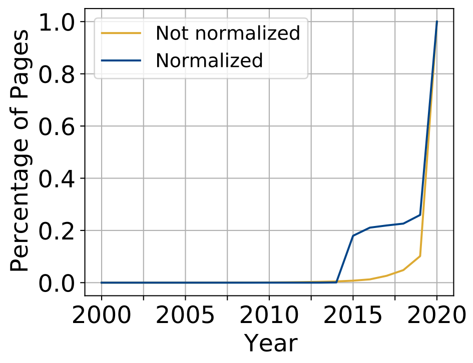
(a) Mean



(b) Median



(c) Boxplot



(d) CDF

Figure 5.1: Analysis of the archive data. **(a)** and **(b)** show the number of pages on a domain, **(c)** shows it as boxplots (yellow markers represent the mean). **(d)** depicts a normalized and non-normalized CDF of which percentages of pages from the current sitemap were present in a specific year.

Interestingly, both CDFs suggests that old pages indeed get removed from a domain eventually. However, the other plots of Figure 5.1 also indicate that sitemaps are growing over the years. This then means that the older an active page is, the less likely it is meant to be still online. We leverage this observation later on in our analysis.

Note that the trends of the graphs rely on the Internet Archive's operations throughout the years. For example, sitemap growth is naturally influenced by the Internet Archive's crawl scale. We elaborate more on this in Section 6.5.

## 5.2. Page Similarity

We identified a first parameter that can provide an indication of whether a page is unintentionally orphaned: its last seen date in the archive. Despite being an indicator, the last seen date of a page is not sufficient on its own to assess the unmaintained status of an orphan pages. Although it increases the possibility, many false positives can still arise if we would blindly follow this attribute.

Fortunately, there is another intuitive indicator of whether a page was intentionally orphaned, namely its similarity to its last seen version. That is, if we can capture the differences between how a page looks now compared to how it looked some years ago, we can get a grasp of the maintenance frequency of this page. Thus, to check whether a page has actually become unmaintained orphaned since its last listing on the Internet Archive, we compare the current version of a page to its last archived version.

Detecting (near-)duplicates among web pages and other types of documents has been extensively studied before, with the state of the art being Broder et al. [12] and Charikar [14]. Henzinger [29] did a comprehensive and large-scale analysis of both, within the context of near-duplicate detection of web pages, and concluded that Charikar's *simhash* achieves better overall precision. Next, Manku et al. [40] proved the potential of Charikar's *simhash* for near-duplicate detection, specifically for the purpose of web crawling. Apart from good precision, *simhash* is fast and incurs a small memory footprint. Thus, we base our similarity detection on it. The procedure works by first creating a fingerprint of two pages, followed by comparing the two fingerprints against each other. The following describes both steps.

### 5.2.1. Creating Fingerprints using simhash

We create the fingerprint of a web page by first removing all HTML tags and extracting only the content of the page. We then divide the extracted content into n-grams of $n = 8$. For each n-gram, we construct a 64-bit hash using the Fowler–Noll–Vo hash function, specifically the FNV-1a hashing algorithm [42]. The hashing algorithm works by choosing a 64-bit **offset basis** and a 64-bit **prime**. This will help us to ensure that each hash produces a 64-bit unsigned integer. We start by assigning our hash-output to the 64-bit **offset basis**. Then, for each *byte* in the input, we do the following two operations:

1. We XOR the lower 8 bits of the hash-output with the current byte of the input. The result becomes the new hash-output.
2. We multiply the new hash-output with the 64-bit **prime** and take the lower 64 bits of the result as the new hash-output.

Algorithm 3 describes the procedure in pseudocode. For our implementation, we use the *pyhash* package in Python, which contains the FNV-1a hash operation (implemented in C) along with its predefined **offset basis** and **prime**.

**Algorithm 3** Fowler–Noll–Vo Hash Function

```
1: $hash \leftarrow$ offset_basis
2: for $byte$ in $input$ do
3:     $hash \leftarrow hash \oplus byte$
4:     $hash \leftarrow hash \times$ prime
```

To construct the fingerprint, we go over each index position $i$ (from 0 up until 63) and sum the value of each hash at that index position in the following way: we initialize our fingerprint to a bitstring of 64 0's. If the hash of the n-gram has the value 1 at index $i$, we add 1 to index $i$ in the fingerprint. If the hash has the value 0, we subtract 1 from index $i$ in the fingerprint. For each index in the fingerprint, we convert the value to 1 if its positive, and to 0 if its negative. The result is a 64-bit fingerprint of the web page. Algorithm 4 describes the procedure in pseudocode.

**Algorithm 4** Fingerprint Creation (*simhash*)

```
1: $ngrams \leftarrow$ List of ngrams on the web page
2: $hashed\_ngrams \leftarrow$ []
3: $fingerprint \leftarrow$ "000...0"                                   ▷ Initialize bitstring of 64 0's
4: for $ngram$ in $ngrams$ do
5:     $hashed\_ngram \leftarrow hash(ngram)$      ▷ Using FNV-1a hashing algorithm, described in Algorithm 3
6: for $hashed\_ngram$ in $hashed\_ngrams$ do
7:     $binary \leftarrow bin(hashed\_ngram)$                                ▷ Convert hash to binary
8:     for $i$ in 0..63 do
9:         $bit \leftarrow binary[i]$
10:        if $bit == 1$ then
11:            $fingerprint[i] \leftarrow fingerprint[i] + 1$
12:        else
13:            $fingerprint[i] \leftarrow fingerprint[i] - 1$
14: for $i$ in 0..63 do
15:     $bit \leftarrow fingerprint[i]$
16:     if $bit > 0$ then
17:         $fingerprint[i] \leftarrow 1$
18:     else
19:         $fingerprint[i] \leftarrow 0$
```

A more visual representation of the procedure is depicted in Figure 5.2. The first step shows how a simple web page is converted into ngrams. Note that, for the purpose of illustration, the ngrams in the figure are ngrams of $n = 2$. The actual implementation uses ngrams of $n = 8$. However, the concept remains the same. We choose $n = 2$ for ease of representation in the figure.

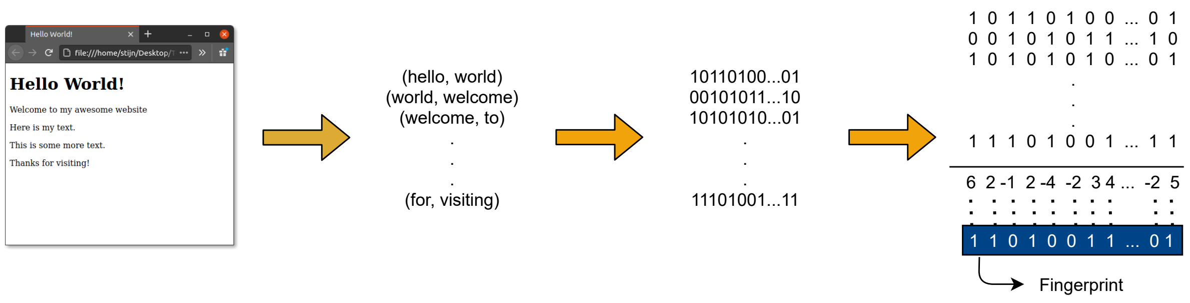


Figure 5.2: Visual representation of the procedure for creating a 64-bit fingerprint for a web page. The example uses ngrams of $n = 2$, while the actual implementation uses ngrams of $n = 8$.

All ngrams are hashed and converted into a 64-bit binary string. Then for each binary string, we add 1 to index $i$ in the fingerprint if the hash is equal to 1 at index $i$, and subtract 1 if otherwise. To obtain the final fingerprint, we convert each index in the fingerprint with a positive value to 1, and to 0 if otherwise.

### 5.2.2. Comparing Fingerprints using Hamming Distance

For each URL in our data set, we retrieve its current version and its latest archived version, and we create fingerprints for both pages. We then compare the 64-bit fingerprints using the Hamming distance, which is the number of bits that need to be changed for the two fingerprints to produce two equal strings. The inverse gives us the similarity.

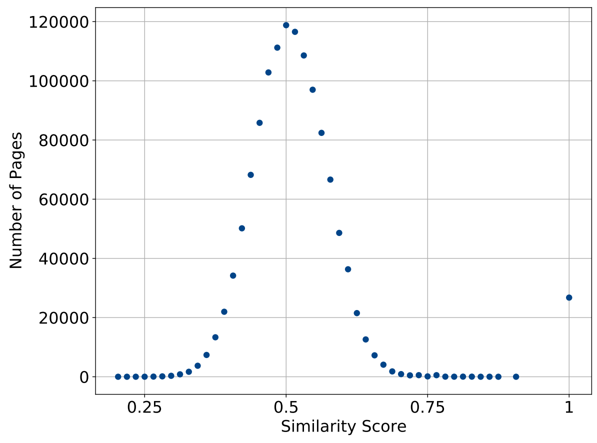


Figure 5.3: Distribution of the calculated similarity scores.

### 5.2.3. Calulating the Similarity of Pages in our Data Set

Figure 5.3 shows the distribution of the similarity scores of the candidate orphaned pages. We observe a clear Gaussian distribution around 0.5 with an increase at 1. Indeed, it follows our expectations that, although pages change over the years (e.g., their design), core content often remains unchanged. The left tail of the curve represents pages that have changed more drastically throughout the years. Their similarity score is lower than average, which means there are many differences between the current version of the page compared to the latest archived version of the page.

The right tail of the curve shows the pages that have remained unchanged. Their similarity score is higher than average, which means there are little differences between the current version of the page compared to the latest archived version of the page. We notice a spike at a similarity score of 1, which signifies the pages that appear to have not changed at all since their latest version in the Internet Archive. While we already suspect these pages to be orphaned (given their missing presence in the latest archive crawls), the high similarity score suggests that these have in fact experienced little to no change over the years. This might again indicate that these pages are forgotten, hence increasing their chance of being orphaned.

## 5.3. Orphan Likelihood Score

We identified two parameters that can provide an indication of whether a page is intentionally or unintentionally orphaned: its last seen date (see Figure 5.1), and its similarity score (see Figure 5.3). We can therefore use these parameters to construct a metric for the likelihood of a page being orphaned. Given that, as a starting point, we took a domain's sitemap from 2020, we can subtract from 2020 the year in which a page was last encountered. We call the result the page's age. From our data, we know that the oldest year a page from our data set was last seen on the archive is 1997. Correspondingly, the maximum age a page in our data set can have is 23 years. The similarity score is normalized to [0-1], and we can easily scale it to [0-23]. As such, we combine the parameters to obtain the orphan score $O$:

$$O = \frac{23Sw_s + Aw_a}{23}$$

Here, $S$ is the similarity score, $A$ is the age (calculated by subtracting the last-seen year from 2020), and $w_s$ and $w_a$ are weights of the similarity score and the age. We divide by 23 to normalize to [0-1].

### 5.3.1. Parameter Evaluation

To determine the values for the weights $w_s$ and $w_a$ we empirically evaluate them on our data set. More precisely, we evaluate values between 0 and 1 for both weights, in increments of 0.1. Concretely, we pick a pair of weights and calculate the orphan score for all pages in our data set. We do so for each possible pair

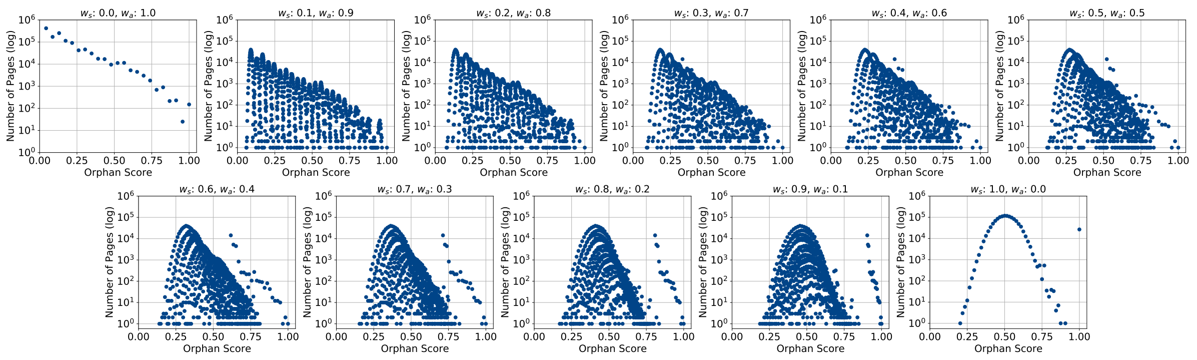


Figure 5.4: Empirical evaluation of the weights in the Orphan Score formula. We see that a higher weight for the similarity score leads to the exclusion of a specific set of links. Chosing the extremes ($w_a = 1$ or $w_s = 1$) leads to a more discrete clustering of the orphan score, thereby increasing the per-score-max.

Figure 5.4 depicts the results. With each increment the distribution of the orphan score slowly excludes a set of links (on the right side of the graph) until it converges to the normal distribution of Figure 5.3 at $w_s = 1$ and $w_a = 0$ (note the log-scale of each graph). The graph becomes split in two: a normal distribution on the left (low orphan score), and a set of outliers on the right of the bell (higher than average orphan score). Using a higher weight for the similarity score allows us to study the set of links that got "detached" from the greater chunk in the graph.

This also makes sense intuitively. While the age plays a crucial role to determine whether a page has been kept up-to-date, it is more prone to false positives. A page can obtain a rather high age without necessarily being orphaned. For example, if a sitemap gets restructured, a page indeed can have a different path from the index page (root node) compared to the previous configuration. While it is not necessarily orphaned, the page can still avoid being detected by the crawler if the new path has a bigger depth in the sitemap graph. This depends both on the depth, as well as the crawler heuristics. As such, it makes sense to put a bigger weight on the similarity score sine it is less prone to false positives. If a page has a certain similarity score, it will always be a direct result of the difference compared to its previous version, which can be a good indicator for maintained or unmaintained web pages.

We therefore use a weight allocation of $w_s = 0.9$ and $w_a = 0.1$, along with a cutoff of 0.9. Meaning, we discard all pages with an orphan score lower than 0.9. In total, we reduce our set of pages to 26,756 URLs.

## 5.4. Classifying the Type of Pages

Following, we classify the identified 26,756 URLs and determine whether they are truly orphaned. We look for specific indicators, in the source code or content, that provides insight into the orphan status. We divide them up in three classes: likely orphaned, not orphaned, and uncertain. With likely orphaned, we identify clear indicators that the page is orphaned, such as outdated copyright statements or incomplete/boilerplate code. Second, we label a page as non-orphaned when we detect that the copyright on the page is at least from 2020. Note, that this will mislabel pages with a dynamically generated copyright, which is in line with our goal of providing a lower bound. Finally, we label a page as uncertain when there is neither a clear indicator for it to be likely orphaned, nor for the page to be non-orphaned. We describe how we detect these indicators as follows:

### 5.4.1. Copyright Statements

Copyright statements, often found at the bottom of a web page, are a way for website owners to protect the content on their website against plagiarism. It indicates to the user of a website that the content falls under copyright law and can hence not be copied without official consent by the website owner. Most often it contains the statement, along with a date and the name of the copyright owner (which is often just the name of the company that owns the website).

Another benefit of copyright statements is that it can give insight into when a page was last updated. In theory, the date of a copyright statement should be updated with every change made to the respective web page. Therefore, if a page mentions a copyright statement from 2017, we can assume the page has not been updated since, and has hence remained unmaintained.

To capture these statements, we scan our pages for the word "copyright" (case insensitive) and the copyright symbol (©). If present, we examine the 50 characters before and 50 characters after the copyright, and aim to detect a date. If the year is 2020 or 2021, we assume the page is up-to-date. If the year is older, we label it as likely orphaned. If no date could be detected, the web page can either be orphaned or non-orphaned, for which we then decide to label the page as uncertain.

It is of course possible that these statements are automatically generated every year. If a page has been unmaintained and forgotten for several years, it might still advertise a copyright statement accompanied with the current year. This would cause us to mistakenly label an orphaned page as non-orphaned, which again stresses our goal of finding a lower bound.

### 5.4.2. Boilerplate Code

Websites often use "boilerplate code" to speed up the development process. They can function as a template complying with the general setup of the website, or as a test page to confirm successful functionality or deployment. This boilerplate code, when rendered, may display an empty page, or a generic (small) set of words. The code itself, however, may be filled with boilerplate HTML tags or JavaScript code blocks, assuring compliance with the general template of the website.

Boilerplate code that is not further developed with content can be an indication of an orphaned web page, presumably because there was intention of deploying an endpoint, which was then later not further pursued, although without removing the page.

We detect these boilerplate web pages by stripping away all HTML-tags and script blocks from the source. If we are left with an empty page, or a page where at most 5 words remain, we flag it as a boilerplate page.

### 5.4.3. Error Pages

We already removed pages that respond with a status code OK (200), yet display an error message (see Section 4.1). Our initial approach required a design that was quick and scalable to handle millions of web pages. We therefore came up with a heuristic that takes the size of a web page, and removes pages from the same domain with a similar size. However, although our heuristic was efficient at-scale, we had to compromise for accuracy. It is therefore possible that error pages with a status code OK (200) are still present in our set. Therefore, on this pre-filtered data set, we now utilize a more expensive and more accurate additional analysis step. To detect these pages, we look for common phrases, such as "Page Not Found." The full list of key-phrases can be found in Appendix B.

### 5.4.4. Following Redirects and Loading Frames

When probing our list of candidate orphan pages, we filter out any page with a status code that is not HTTP OK (200) response. This includes the range of codes that denote a 3xx redirect. While effective at-scale, this approach misses specific redirects, for example in JavaScript or HTML meta tags. We detect these redirects by comparing them against a list of known in-page redirect techniques (see Appendix B). Similarly, we can use a list of HTML tags used for loading frames to detect these types of pages. Albeit pages making use of redirects or frames can be classified as such, they might still be unmaintained or forgotten. Below we describe our method for detecting and processing the two cases:

**Following Redirects:** As mentioned before, we observe two ways in which a page can redirect itself. The first, making use of HTTP, is easy to detect via the status code in the HTTP response header. We filtered earlier for this by specifically looking for response codes in the 3xx range, and discarding any page responding with such code. A second way of inflicting redirects is through code or HTML meta tags. These are less straightforward to detect at-scale as it requires analyzing the source code of a page, and comparing it against a dictionary of common redirect techniques. This causes a significant overhead, making it unfit for large-scale deployment. Fortunately, we have pruned our data set as such that we can afford more rigorous detection mechanisms at this stage in the pipeline.

We classify in-code redirects as follows: For each page, we analyze the source code and compare it against a list of common strings found in redirect commands (see Appendix B). In case of a match, we try to parse the redirecting-code and extract the URL the page needs to redirect to. We manually follow the redirect and perform the same analysis as described above (copyright, boilerplate, etc.) on the redirected-to page. In case another redirect is encountered (or frame, see below) we repeat the process. We might encounter a loop, that is a page redirects to another redirecting page, possibly resulting in an infinite redirect loop. Like modern browsers, we limit the number of consecutive redirects. In our case, we follow a maximum of 20 redirects.

**Loading Frames:** In HTML you can use frames to split the layout of your web page and individually load the content of each frame from a specific source file. Especially with the increase of mobile devices, frames can cause issues when handled from a tablet or smartphone, making them less ideal than initially thought of. As such, the World Wide Web Consortium (W3C) have deemed them obsolete in HTML 5 because it *“damages usability and accessibility”* [15]. Nonetheless, we still encounter them often when classifying our web pages (which, again, can also be an indication on its own of how outdated/unmaintained a web page is).

Because each frame loads their content from a separate source file, we cannot directly parse (and hence, classify) the content on the page. Although we visually see the content when rendering the source code, we cannot analyze it programmatically since the code is dynamically inserted. We therefore need to process the frame tags such that we can download and parse the actual code, which then allows us to properly classify the web page.

We apply a similar process for handling frames as we did with following redirects. We analyze for each page the source code and compare it against a list of common strings found as frame-tags (see Appendix B). If there is a match, we parse the frame tags and try to extract the URL of a source file. In case of multiple source files, we pick the first and continue the process. Once a URL is obtained, we load the source file and apply again the same classification as described above.

## 5.5. Removing One Domain Name

When analyzing our remaining 26,756 pages we notice one domain name evading DUDe as it was spread over multiple TLDs, biasing our results with a little over 20,000 entries. We missed these particular cases because the pages are spread over multiple TLDs (`.my`, `.hk`, `.tw`, `.net`, `.ie`, `.ae`, and `.id`), and our heuristic functions on a per-domain basis (domain name + TLD).

Remarkably, the TLDs of the domain are primarily Asia-oriented, with the exception of `.net` and `.ie`. We manually filter out the pages using this SLD to have a more diverse set of domains that gives a better depiction of the accuracy of our methods. Doing so leaves us with 5,914 pages.

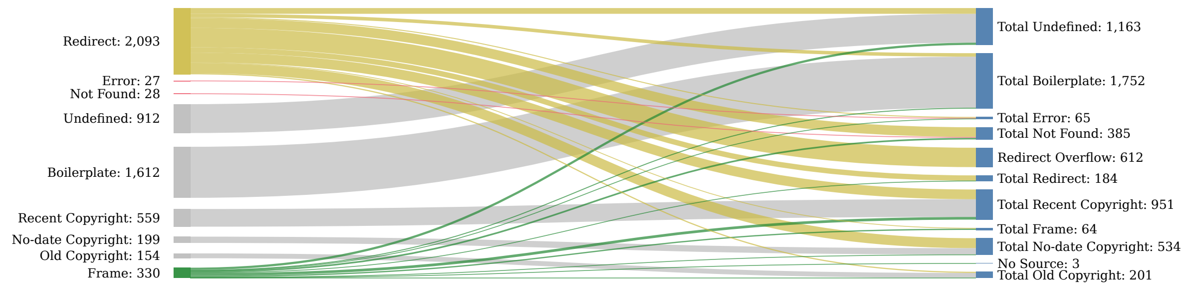


Figure 5.5: Sankey diagram of the detected page types. On the left we have the initial classification. The flows in the diagram depict the result of following redirects and loading frames, which leads to the results of our classification shown on the right.

## 5.6. Analyzing Types of Pages

For all links left in our set, we download their latest version as of March 14, 2021 and analyze them for the type of pages (redirects, boilerplate, etc.) as described above. Figure 5.5 shows a Sankey diagram of the results. On the left, we see our initial classification. On the right, we have the classification after following redirects and loading frames. The flows in the diagram depict how many of the redirects and frames could be classified in one of the other categories, therefore showing the value of following redirects and loading frames.

### 5.6.1. Classifying the Results

The most common pages we encounter are in-page redirects via HTML or JavaScript (2,093 pages). Although redirecting can be a mechanism to divert traffic from orphaned pages, our results indicate that many of these redirects tend to be misconfigured (45%), of which the majority seems to cause an endless loop of redirects (65% of the misconfigured redirects). From the redirecting links that end up at a page with a copyright statement, we see around an equal amount with a recent timestamp and without (both 17%).

Second most common, we see the boilerplate pages (1,612 initial pages, 1,752 after redirects and frames). We believe these are forgotten. We assume there was initial intent to deploy content, which was then not further pursued afterwards. This could be due to a myriad of reasons, such as simply forgetting about it or refactoring.

Particularly interesting are the pages that contain a copyright statement. In total, we found 1,686 such statements, with the majority of the pages actually having an up-to-date timestamp (56%). While these might be dynamically generated, we consider these pages maintained in the spirit of identifying a lower bound. The ones that do not have an up-to-date timestamp are divided into two groups: statements with a timestamp (12%), and statements without any timestamp (32%). When a timestamp is present, we assume the page was last maintained during that year. These pages are likely orphaned, while the copyright statements without a timestamp could be orphaned, but might not.

Figure 5.6 depicts the distribution of the years extracted from copyright statements. The majority of the pages seem to be last maintained in the past 5-10 years. However, we also observe some pages containing a copyright from the year 2000, which suggests these orphaned web pages have not been touched in more than 20 years. Although the majority of the copyright years were found during our initial classification, many redirects eventually lead to an outdated orphan page as well. This shows that following such redirects, as well as loading frames, does indeed reveal more orphan pages and are therefore not per se a sign of a maintained web page.

**Answering the Research Question**

(**RQ2.1**): *How do orphan pages look like from a programming perspective?*
From our classification, we can conclude that many pages appear to be standard boilerplate templates, presumably used to assure compliance with the rest of the web application, or to verify successful deployment. We also observe many pages making use of frames, indicating their old (and potentially unmaintained) status since frames have been deemed obsolete in HTML5 [15]. Interestingly, a large part of the pages function as a redirect. However, when following though on these redirects we notice nonetheless orphaned or misconfigured content.

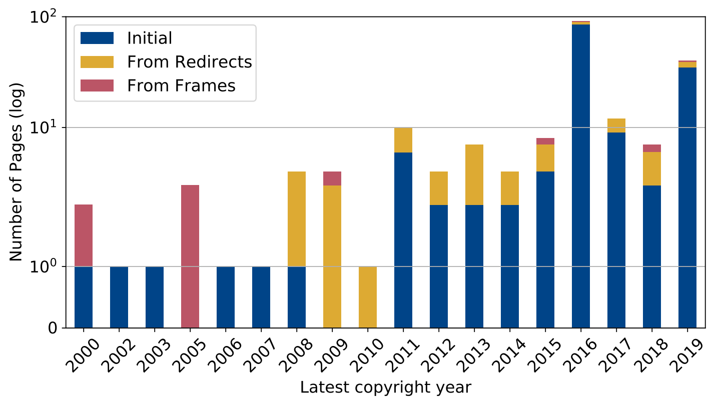


Figure 5.6: Extracted copyright years from copyright statements on pages that are not considered to be up-to-date. The majority of the pages were last touched at some point in the past 5-10 years, although we also observe pages which appear to be unmaintained for much longer.

### 5.6.2. Impact on Orphan Lower Bound

Table 5.1 summarizes the types of pages and their orphaned status. For at least 1,953 URLs, the pages appear forgotten, unmaintained, and therefore likely orphaned. These pages are spread over 907 domains, with one domain having as many as 142 likely orphaned pages. For 1,706 pages, the decision is not as clear. While we could not detect any clear sign of being maintained, we also do not know if they are not. Though, intuition leads us to believe that at least some set of these pages are most likely unmaintained as well. Lastly, we identify 951 pages which appear maintained and not orphaned.

With this analysis, we have captured the first lower bound on orphan web pages in the wild. This proofs two things:

1. Our methodology works for detecting orphaned web pages in the wild and at-scale. Due to its modular build and fit for parallelism we were able to smoothly deploy our implementation for large-scale measurements without any issue.

2. Orphan web pages are in fact present in the wild and we have seen one domain having as many as 145 likely orphaned pages. Although most of them appear to be less than 10 years old, we do encounter pages that seem unmaintained for at least 20 years.

We elaborate more on these two points, as well as other results from this study, in our discussion in Chapter 6.

Table 5.1: Summary of the most important types of pages. We observe that our method is capable of detecting web pages that are likely to be orphaned, and we can say that 951 pages are in fact not orphaned. We also see a large group of pages for which the determination is not clear.

| **Type** | **Count** | **Orphan Status** | **Total** |
|---|---|---|---|
| Old Copyright | 201 | Likely Orphaned | 1,953 |
| Boilerplate | 1,752 | | |
| Copyright without date | 534 | Uncertain | 1,706 |
| Undefined | 1,163 | | |
| Recent Copyright | 951 | Not Orphaned | 951 |

**Answering the Research Question**

(**RQ1.3**): *What is a lower bound on the number of orphaned web pages in the wild?*
Using our designed methodology from Chapter 3 in a month-long large-scale measurement study on 100,000 domains, we identify at least 1,953 pages, spread over 907 domains, as orphaned.

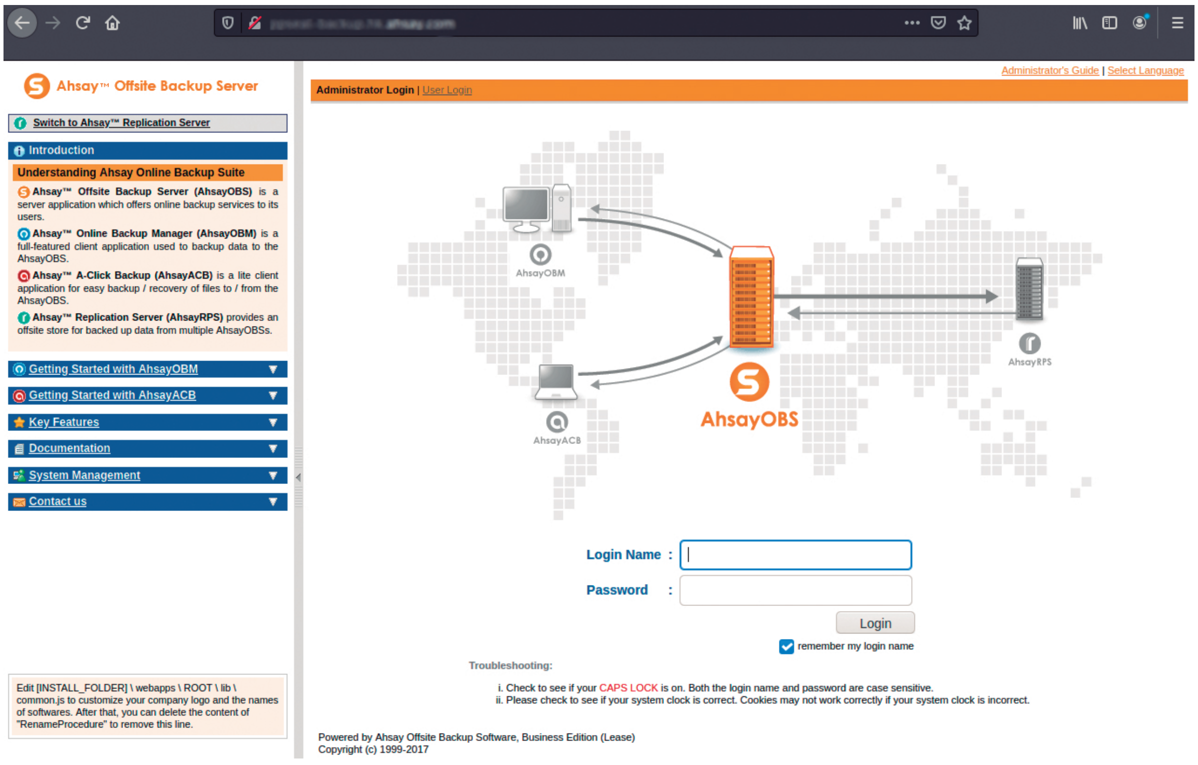
### 5.6.3. Orphan Page Examples

We present two examples of orphaned web pages in Figure 5.7 and Figure 5.8. Figure 5.7 shows what seems to be an (outdated) login panel to a company's offsite backup server. The company name is incorporated as a subdomain in the URL (which is blurred for privacy purposes). The bottom of the page shows how the page was last stamped by a copyright in 2017, suggesting the page has been left unmaintained for at least 4 years.

In Figure 5.8 we can again observe an (outdated) login panel. This time, the page seems to be able to give access to a general purpose application server which, as mentioned below on the page, would function to assist a logged in user in *"dealing with Sony and customer's products and accessories."* Note again how the copyright dates from 2017, suggesting again that the page has been left unmaintained for at least 4 years.

Although equipped with an authentication portal, both URLs show an example of outdated pages that pose a real threat if an adversary would obtain access to them. Especially given their outdated nature, we have fewer guarantees that the login portals are sufficiently protected along modern-day security standards. Regardless of the direct security threats, however, these pages can assist adversaries with other attacks. Knowing their existence and visual aspects, it is much easier for an attacker to create a realistic phishing page based on these orphaned web pages. As such, orphan web pages can pose a threat to the security of a web application if they disclose information useful for an attackers as *Open Source Intelligence* (OSINT).

We analyze the security of all found orphan web pages later on in this chapter. Also in Chapter 6 we elaborate more on different orphan page examples, along with their impact on security.


Ahsay™ Offsite Backup Server

Switch to Ahsay™ Replication Server

Introduction

Understanding Ahsay Online Backup Suite

**Ahsay™ Offsite Backup Server (AhsayOBS)** is a server application which offers online backup services to its users.

**Ahsay™ Online Backup Manager (AhsayOBM)** is a full-featured client application used to backup data to the AhsayOBS.

**Ahsay™ A-Click Backup (AhsayACB)** is a lite client application for easy backup / recovery of files to / from the AhsayOBS.

**Ahsay™ Replication Server (AhsayRPS)** provides an offsite store for backed up data from multiple AhsayOBSs.

Getting Started with AhsayOBM

Getting Started with AhsayACB

Key Features

Documentation

System Management

Contact us

Edit [INSTALL_FOLDER] \ webapps \ ROOT \ lib \ common.js to customize your company logo and the names of softwares. After that, you can delete the content of "RenameProcedure" to remove this line.

Administrator's Guide | Select Language

Administrator Login | User Login

Login Name :

Password :

Login

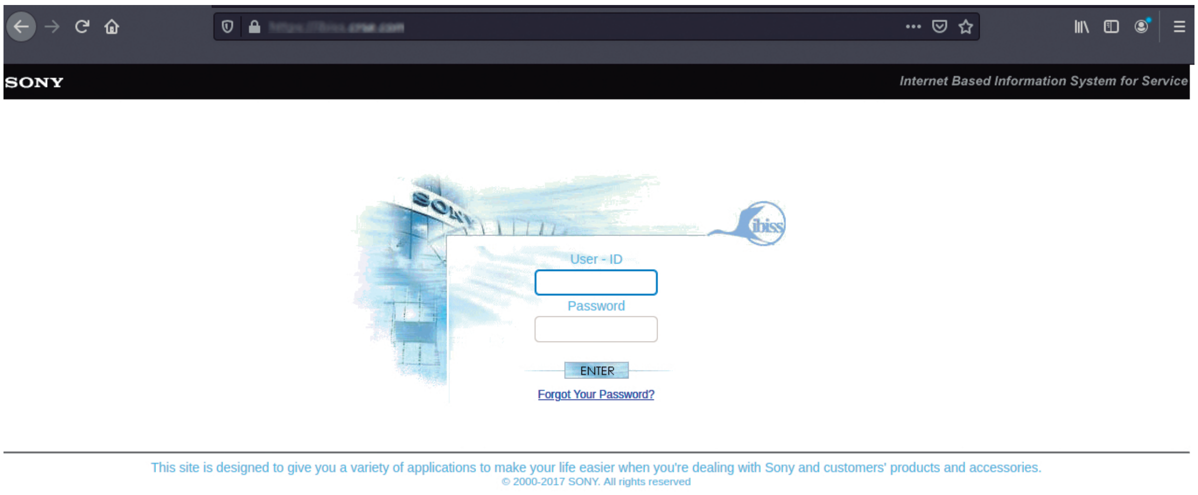
[✓] remember my login name

Troubleshooting:

i. Check to see if your CAPS LOCK is on. Both the login name and password are case sensitive.
ii. Please check to see if your system clock is correct. Cookies may not work correctly if your system clock is incorrect.

Powered by Ahsay Offsite Backup Software, Business Edition (Lease)
Copyright (c) 1999-2017

Figure 5.7: Example of an orphaned web page. The page seems to be a login panel for a company's offsite backup server. Note how the copyright at the bottom is from 2017.



This site is designed to give you a variety of applications to make your life easier when you're dealing with Sony and customers' products and accessories.
© 2000-2017 SONY. All rights reserved

Figure 5.8: Example of an orphaned web page. The page seems to be a login panel for a company's general purpose application server. Note how the copyright at the bottom is from 2017.

## 5.7. Google Visibility of Orphaned URLs

Before analyzing the security status of orphaned web pages, we take a look at how easy it is to find these pages through a search engine. Given their orphaned status, we suspect these pages are not easily caught by web crawlers. Depending on how often a search engine updates its index, orphan page will gradually disappear from search engine databases and will, hence, become more cumbersome to find. We take Google as a case study example to test whether or not we can find these pages by making use of common search engines.

### 5.7.1. Google Dorks and the Programmable Search Engine

To test how many of our pages are indexed by google we need two things: 1) a precise and unique query that only matches the web page we are looking for, and 2) a scalable way to launch this query on the Google search engine. Fortunately, Google has the proper tools in place to achieve both through the form of Google Dorks and the Google Programmable Search Engine. We explain both briefly:

**Google Dorks:** Apart from querying general search terms, Google allows users to perform a more advanced search to look for specific file types or to provide certain structure characteristic of a website [23]. For example, if one were to look for a master thesis, one can accompany the search query with the `filetype` operator to look only for PDF files in the following way: "`master thesis filetype:pdf`". This will return all search results for the term "master thesis" that are also PDF files.

Although no formal definition exists yet, Google Dorks are generally defined as queries taking advantage of Google's refined search terms, often with the goal of finding sensitive or vulnerable information on a domain [58]. For example, using the same operator as in the previous example, one could construct a Google Dork to look for publicly stored password files in the form a Excel sheets. The Dork would look something like this[1]: "`passwords filetype:xls`".

For our particular use case (which is looking for indexed orphan pages) we need a query that satisfies two requirements:

1. If the orphan pages is indexed by Google, the query should return at least one result.
2. If the orphan page is not indexed by Google, the query should return no results.

If these requirements are met, we can use the query for each URL in our orphan list, and get a binary answer back (1 for results, 0 for no result) to see if the page is indexed by Google. We achieve this by using the `site` operator, which will only return results residing on the provided URL. For example, querying "`phone number site:tudelft.nl`" will return pages for the term "phone number" that are also on `tudelft.nl`. If we provide the URL of our orphan page along with the `site` operator, we will get all results for that specific

[1] Example given for educational purposes only.

URL. Therefore, if the URL is indexed we get at least one result. If the URL is not indexed, we get zero results. This satisfies our requirements for the query.

**Google Programmable Search Engine:** To allow web developers to integrate high-quality search engine results on their website, Google has designed the Programmable Search Engine (formerly known as Google Custom Search) [24]. The Programmable Search Engine makes use of Google's core search technology, meaning that website administrators can allow for tailored search queries supported by Google. Furthermore, it allows website owners to make additional revenue by incorporating advertisement from Google AdSense to their search queries, and to analyze user behavior using Google Analytics.

To make use of the Programmable Search Engine, a user needs to create a custom instance of the engine (which sort of functions as the user's own 'personal search engine', although powered by Google), after which queries can be sent to the created search engine. The results of the queries will depend on the user's own configuration of the Programmable Search Engine. Convenient for our research is that each custom instance of a Programmable Search Engine can be accessed via an API [22], which therefore allows us to perform queries in bulk, and avoids the need for parsing regular Google searches from HTML. We set up our own Programmable Search Engine, and configure it to work as the regular Google search engine. Making use of the custom instance works by making a request to:

```
https://www.googleapis.com/customsearch/v1?key=[API-KEY]\&cx=[CX-KEY]\&q=[QUERY].
```

where `[API-KEY]` is our personal API key, `[CX-KEY]` is the identifier of our Programmable Search Instance, and `[QUERY]` is the query we want to search for in the Google search engine.

The response we get is a JSON string containing the results of our search query. Listing 5.1 shows an example of such response for querying "`site:https://www.tudelft.nl/cybersecurity/focus-areas/network-security`". Because of the specific URL, we get exactly one response back from the search engine, which can be easily parsed from the JSON entry '`totalResults = 1`' on line 28 in the listing[2].

### 5.7.2. Finding Indexed Orphan Pages

We make use of Google's Programmable Search Engine to determine how many of our pages are indexed by Google. We construct a simple query (Google Dork) that searches for "`site:<url>`" and count the number of results returned by Google. If no results are returned, we assume the page is not indexed by the Google search engine. If at least one result is found, we know that the web page is indexed by Google

We perform the lookups between March 10, 2021 and March 12, 2021 (spread over 3 days due to rate limiting) on all 5,914 URLs. We discover that, in total, only 16% of the pages were indexed by Google. This shows that many of our (likely) orphaned web pages are not (easily) reachable through the web, and hence difficult to find without making use of archived databases. This is an important finding, as it demonstrates the need for our technique. Presumably because search engines perform regular updates of their indexing database, it is to be expected that over time an orphan page has fewer chance to be retrievable by search engines. It stresses again the fact that, without knowing the specific URL of an orphan page, these pages are extremely difficult to find without making use of archive databases.

For completeness, we analyze the indexed pages further to see if we can find any pattern among the types we classified earlier. This is not the case; we observe no correlation between any of the identified types (boilerplate, redirect, etc.) and the indexed status of a page, neither between any of the orphaned statuses (likely orphaned, uncertain, etc.) and Google's indexing. Only pages for which we detected an old copyright statement we see that the more recent a page, the more chance of it being indexed. However, we expect this is just following the general distribution from Figure 5.6, or that they will soon be removed by Google.

## 5.8. Security Status of Orphan Pages

Lastly, we take a look at the vulnerability of orphan web pages. We have shown that the pages in our set appear to have no ancestry link anymore within the sitemap of their website. Furthermore, we found evidence that a large portion of these pages have not been maintained for a while, or still contain standard boilerplate code from many years back. Because of their unmaintained status and relatively old age, we assume these pages have not been kept up-to-date with the current security standards, which can make them more vulnerable to security exploits. To test this assumption, we run a security scan on the found orphan pages, and compare the results against a control group.

[2]Note that a similar entry is present on line 11.

```
{
"kind": "customsearch#search",
"url": {
"type": "application/json",
"template": "https://www.googleapis.com/customsearch/v1?q={searchTerms}&num={count?}&start={
    ↝ startIndex?}&lr={language?}&safe={safe?}&cx={cx?}&sort={sort?}&filter={filter?}&gl={gl?}&cr={
    ↝ cr?}&googlehost={googleHost?}&c2coff={disableCnTwTranslation?}&hq={hq?}&hl={hl?}&siteSearch={
    ↝ siteSearch?}&siteSearchFilter={siteSearchFilter?}&exactTerms={exactTerms?}&excludeTerms={
    ↝ excludeTerms?}&linkSite={linkSite?}&orTerms={orTerms?}&relatedSite={relatedSite?}&
    ↝ dateRestrict={dateRestrict?}&lowRange={lowRange?}&highRange={highRange?}&searchType={
    ↝ searchType}&fileType={fileType?}&rights={rights?}&imgSize={imgSize?}&imgType={imgType?}&
    ↝ imgColorType={imgColorType?}&imgDominantColor={imgDominantColor?}&alt=json"
},
"queries": {
"request": [
        {
        "title": "Google Custom Search - site:https://www.tudelft.nl/cybersecurity/focus-areas/
            ↝ network-security",
        "totalResults": "1",
        "searchTerms": "site:https://www.tudelft.nl/cybersecurity/focus-areas/network-security",
        "count": 1,
        "startIndex": 1,
        "inputEncoding": "utf8",
        "outputEncoding": "utf8",
        "safe": "off",
        "cx": "XXXXXXXXXXXXXXXXX"
        }
]
},
"context": {
        "title": "Google"
},
"searchInformation": {
        "searchTime": 0.28269,
        "formattedSearchTime": "0.28",
        "totalResults": "1",
        "formattedTotalResults": "1"
},
"items": [
        {
        "kind": "customsearch#result",
        "title": "Network Security",
        "htmlTitle": "Network Security",
        "link": "https://www.tudelft.nl/cybersecurity/focus-areas/network-security",
        "displayLink": "www.tudelft.nl",
        "snippet": "Networks connect almost all 6 billion inhabitants of planet earth. You can phone
            ↝ , \nchat, or mail friends and family wherever they are, at almost no cost. But an
            ↝ ...",
        "htmlSnippet": "Networks connect almost all 6 billion inhabitants of planet earth. You can
            ↝ phone, \u003cbr\u003e\nchat or mail friends and family wherever they are, at almost
            ↝ no cost. But an&nbsp;...",
        "cacheId": "9v6VpAAoUqEJ",
        "formattedUrl": "https://www.tudelft.nl/cybersecurity/focus-areas/network-security",
        "htmlFormattedUrl": "https://www.tudelft.nl/cybersecurity/focus-areas/network-security",
        "pagemap": {
                "cse_thumbnail": [
                        {
                        "src": "https://encrypted-tbn1.gstatic.com/images?q=tbn:ANd9GcR49oqmQZ5
                            ↝ MihhYkAWVC-4E_Mh46bEYbgBo2Y7DE4Y1n5PCPPQwP4vmf1A",
                        "width": "225",
                        "height": "225"
                        }
                        ],
                "metatags": [
                        {
                        "msapplication-tilecolor": "#2d89ef",
                        "og:image": "https://www.tudelft.nl/typo3conf/ext/site_tud/Resources/Public/
                            ↝ Images/OpenGraph/TUDelft.jpg",
                        "msapplication-config": "/typo3conf/ext/tud_styling/Resources/Public/img/
                            ↝ browserconfig.xml",
                        "twitter:card": "summary",
                        "og:type": "website",
                        "theme-color": "#ffffff",
                        "og:site_name": "TU Delft",
                        "viewport": "width=device-width, initial-scale=1.0",
                        "og:title": "Network Security",
                        "msapplication-tileimage": "/typo3conf/ext/tud_styling/Resources/Public/img/
                            ↝ mstile-144x144.png"
                        }
                        ],
                "cse_image": [
                        {
                        "src": "https://www.tudelft.nl/typo3conf/ext/site_tud/Resources/Public/
                            ↝ Images/OpenGraph/TUDelft.jpg"
                        }
                        ]
                }
        }
        ]
}
```

Listing 5.1: JSON response from querying the Programmable Search Engine API. To avoid public use of our custom instance we anonymize the “cx” field.

### 5.8.1. Vulnerability Scanning

We use Wapiti [54] to perform a rudimentary security audit of our orphan pages. Wapiti is a web-application vulnerability scanner that looks for common vulnerabilities in web pages. It uses a black-box approach which means that the scanner does not do any analysis test on the source, but instead interacts with the live web page to discover vulnerabilities. To find these vulnerabilities, Wapiti makes use of a technique called 'fuzzing', which entails that the scanner will attempt to inserts specific unwanted payloads that will get triggered when a vulnerability is present (without actually exploiting that vulnerability to cause damage or harm to the website, see Section 6.4 for a discussion on the ethics of our approach).

We focus on two particular exploits: Cross-Site Scripting and SQL Injection. We explain both briefly:

**Cross-Site Scripting:** With Cross-Site Scripting (XSS) an adversary injects and executes arbitrary JavaScript code into a web application due to unsanitized user-input [63]. Concretely, when a web application accepts user input for one of its functionalities (e.g. placing a comment on a forum post or performing a search query on the website) an adversary can enter JavaScript code instead. If the input is not sanitized (that is, checked whether code is being inserted) the web application will interpret it as regular JavaScript code, hence executing the adversaries commands.

Listing 5.2 shows an example of a code snippet that is vulnerable to an XSS-attack. We see that the input variable `searchterm` is not sanitized, which means any code provided as input will be blindly executed by the JavaScript engine. Because the input is taken from the URL, an attacker can easily inject code into the URL and send it to its victim:

`www.example.com/vulnerable-page?q=[MALICIOUS JAVASCRIPT]`

If a website gives each logged-in user a session ID cookie, the following steps could be taken by an attacker to steal that cookie using XSS:

1. An attacker injects malicious JavaScript into the URL of the website (as user input). The JavaScript code is a simple function that retrieves the session ID cookie and sends it to the attacker's server.
2. The attacker sends the URL, containing the malicious JavaScript, to the victim.
3. If the victim opens the URL, the malicious code will be taken from the input and executed by the JavaScript engine. The code hence takes the session ID cookie and sends it to the attacker's server.
4. Once the attacker obtains the victim's session ID cookie from the malicious server, she can log in to that website, impersonating the victim and accessing all his data.

```
<html>
  <body>
    <p id="searchterm"></p>
    <script>
    searchterm = location.href.split("q=")[1];
    searchterm = decodeURIComponent(searchterm);
    document.getElementById("searchterm").innerHTML = "You searched for: " + searchterm;
    </script>
  </body>
</html>
```

Listing 5.2: Code snippet vulnerable to Cross-Site Scripting [13].

Much work has been done on the detection and prevention of XSS vulnerabilities [63, 56, 53, 35, 52, 51]. Also from the industry, we have seen many endeavors to bring caution to XSS attacks, such as the OWASP Top 10 [21], or companies like Secure Code Warrior which are aimed at training developers to write secure code through the means of gamification [64]. Nonetheless, XSS still remains a prevalent vulnerability among web applications online, making it an attractive attack vector for adversaries [51].

Albeit first released in 1995, JavaScript only became widely adopted with the surge of available libraries and AJAX, such as the prominent jQuery library, which was first released in late 2006 and is still being widely used today. Encountering JavaScript, and therefore XSS vulnerabilities, before 2007 is much less likely, which is why we focus on pages from 2007 or later.

**SQL Injection:** Structured Query Language (SQL) is a language designed for accessing and manipulating data, typically stored in a relational database management system (RDBMS). It is commonly used by web applications and various other types of software due to its easy to read and relatively straightforward syntax. Tasks such as verifying user credentials or retrieving account information are one of the many operations often achieved by predefined SQL queries. If user input is inserted into such a predefined query without input sanitation, an adversary is able to inject SQL code into the query, potentially accessing and manipulating the database, which can contain sensitive (user) information.

Listing 5.3 shows an example of a code snippet vulnerable to an SQLi. The code functions as a login form to the web application. Line 8 contains the query, in which user input gets inserted via the variables `username` and `password`. As with the example of the XSS-attack, we see that user input is not validated, allowing the adversary to inject SQL code. In our example, there are many ways the input can be manipulated to bypass the login validation. The most straightforward would be to provide as user input for the variable `username` the string: '`" OR 1 = 1 #`' (without single quotes). Once inserted into the query, the query becomes:

`SELECT * from users where username="" or 1 = 1 # and password = ""`

This query will always evaluate to true, hence yielding a successful login into the web application for the adversary. Note that, because the # operator signifies a comment in SQL, it does not matter what is provided as the `password` input since it will be commented out no matter what. As such, it suffices to leave that field empty in the input parameters.

Also SQL Injections have been extensively covered by the literature [6, 31, 27, 47, 34]. Just as with XSS-attacks, it is prominently featured by the OWASP Top 10, yet, remains very much present in many web application on the Internet [31].

```
1  <h1>natas14</h1>
2  <div id="content">
3  <?
4  if(array_key_exists("username", $_REQUEST)) {
5    $link = mysql_connect('localhost', 'natas14', '<censored>');
6    mysql_select_db('natas14', $link);
7
8    $query = "SELECT * from users where username=\"".$_REQUEST["username"]."\" and password=\"".
         ↪ $_REQUEST["password"]."\"";
9    if(array_key_exists("debug", $_GET)) {
10     echo "Executing query: $query<br>";
11   }
12
13   if(mysql_num_rows(mysql_query($query, $link)) > 0) {
14     echo "Successful login! The password for natas15 is <censored><br>";
15   } else {
16     echo "Access denied!<br>";
17   }
18   mysql_close($link);
19 } else {
20 ?>
21
22 <form action="index.php" method="POST">
23 Username: <input name="username"><br>
24 Password: <input name="password"><br>
25 <input type="submit" value="Login" />
26 </form>
27 <? } ?>
28 <div id="viewsource"><a href="index-source.html">View sourcecode</a></div>
29 </div>
30 </body>
31 </html>
```

Listing 5.3: Code snippet vulnerable to SQL Injection taking from the OverTheWire Natas14 challenge [43].

### 5.8.2. Evaluating Orphan Page Security

We use Wapiti [54] to scan for XSS and SQLi vulnerabilities. Before doing so, we need to define a control experiment which will allow us to contextualize the results. Concretely, we want to compare the scan reports from the orphan pages against similar reports for non-orphan pages. Rather than relying on the literature (which almost certainly not have the same setup as we do), we construct our own control group and perform the exact same scan on the URLs of that group. To do so, we pick 2,000 random pages from our archive data that were encountered by the Internet Archive in 2020. We are assured that these pages have been recently crawled, and are, hence, not orphaned. Table 5.2 summarizes our findings.

We find that 7.1% of the detected orphaned web pages are vulnerable to XSS. They are spread over 13 domains. Approximately 15% of these pages are labeled as boilerplate by our earlier classification. In our control group, we detect XSS vulnerabilities in only 0.9% of the pages, showing that orphaned web pages are *much more* likely (7.9x) to be vulnerable against this type of attack ($p < 0.01$ using $\chi^2$).

Concerning SQLi attacks, our scan reports that 1.3% of orphaned pages are vulnerable. This happens exclusively on boilerplate pages. Compared to our control group, we see no significant difference in the number of pages containing SQLi vulnerabilities.

Table 5.2: Prevalence of Cross-Site Scripting (XSS) and SQL Injection (SQLi) vulnerabilities among orphan pages and control group. For XSS, we only consider orphan pages from 2007 and later, after the first release of jQuery, and a general surge of client-side JavaScript adoption.

| | **Control** | **Orphaned** | | $p$ ($\chi^2$) |
|---|---|---|---|---|
| XSS | 17/1875 (0.9%) | 128/1812 (7.1%) | Boilerplate: 20/128<br>Non Boilerplate: 108/128 | <0.01 |
| SQLi | 23/1875 (1.2%) | 25/1953 (1.3%) | Boilerplate: 25/25<br>Non Boilerplate: 0/25 | ≈0.88 |

**Answering the Research Question**

(**RQ2.2**): *How many orphan pages are susceptible to a Cross-Site Scripting vulnerability?*
Using black-box security scans performed by Wapiti [54], we find that 7.1% of our orphaned web pages are vulnerable to Cross-Site Scripting.

**Answering the Research Question**

(**RQ2.3**): *How many orphan pages are susceptible to an SQL Injection?*
Using black-box security scans performed by Wapiti [54], we find that 1.3% of our orphaned web pages are vulnerable to SQL Injection.

Additionally, we summarize minor security issues (also found by Wapiti), such as missing secure cookies or HTTP Secure Headers, in Table 5.3. Again, we can observe how orphaned web pages tend to be less secure compared to our control group. Especially on Content Security Policy (CSP) and HTTP Secure Headers we see again a significant more issues in orphaned web pages (both $p < 0.01$ using $\chi^2$). We discuss both issues briefly.

**Content Security Policy:** Content Security Policy (CSP) is a concept originally developed by Mozilla with the goal of providing an additional layer to the security of web applications [50]. With a CSP, web administrators are able to put restrictions and requirements on how content should interact on the website. This can help to detect and prevent attacks such as XSS, which we have discussed earlier in this section. A CSP is defined and set by the web application server in the HTTP header. Upon receipt, the web browser then enforces the rules during the execution of the web application.

CSPs are not designed to function as a primary defense mechanism (which is why we consider them as a 'minor' issue). Rather, they should be seen as one of many layers in a website's defense strategy, making it more difficult for adversaries to provide malicious payloads to an application. Nonetheless, failing to have adequate CSPs in place can still lead to a website getting compromised by, for example, an XSS-attack.

From our Wapiti scan, we observe that almost all orphaned web pages (97.1%) fail to have a proper CSP in place. Although this issue is also widely present in non-orphaned web pages (93.8%), orphaned web pages still can be considered significantly worse ($p < 0.01$ using $\chi^2$) when it comes to CSP.

**HTTP Secure Headers:** Similar to CSP, there are various other policies that can be enforced through HTTP Headers. Generally, the standard HTTP Security Headers offer less configuration options than CSP and rely therefore more on built-in heuristics to protect against, for example, XSS-attacks. Although there exist many HTTP Headers that achieve almost identical results as a CSP, it is still recommended to have both enabled for your website. Main reason being that CSP is not often supported by older (versions of) web browsers.

It is therefore no surprise that we see very similar trends in our vulnerability scans for HTTP Secure Headers compared to our scans for CSP. Again, nearly all orphaned web pages seem to have shortcomings when it comes to defining these headers (95.2%), which appears less often in our control group (88.6%). The results are again considered significant, due to a $p$-value of $< 0.01$ using $\chi^2$.

Table 5.3: Comparison of minor security issues in a control group and orphaned pages. We omit $p$-values for path traversal and open redirects due to the too small effect size.

| | **Control** n=1875 | **Orphaned** n=1953 | $p$ ($\chi^2$) |
|---|---|---|---|
| No Content Security Policy (CSP) | 1758 (93.8%) | 1896 (97.1%) | <0.01 |
| Missing HTTP Secure Headers | 1660 (88.6%) | 1859 (95.2%) | <0.01 |
| No Secure Flag Cookie | 738 (39.4%) | 644 (33.0%) | <0.01 |
| No HttpOnly Flag Cookie | 545 (29.1%) | 578 (29.6%) | ≈0.72 |
| Path Traversal | 0 (0%) | 2 (<1%) | - |
| Open Redirect | 1 (<1%) | 1 (<1%) | - |

### 5.8.3. Outdated Software

Finally, outdated software versions could indicate the vulnerability of a web page if a CVE exists for the software version in use. Having already retrieved the source code for our orphan pages, we perform a static analysis to detect a <major>.<minor>.<revision> pattern to see if we can track old software. However, only 8% of the pages contained such a version pattern, most of which being boilerplate pages listing the software they were generated with. Most popular is IBM WebSphere, and one particular page using Claris Home Page, which had its final release in 1998.

# 6 Discussion

In this chapter, we discuss the results and impact of our work. Based on the security analysis of orphaned web page in the previous chapter we elaborate more on the implications of security issues in these web pages, and how they differ from security issues in non-orphaned web pages. This allows us to answer the final research question **(RQ2.4)**.

Furthermore, we cover and explain the applied and scientific use cases for our methodology. We consciously made our implementation open source allowing others to either make use of our technique, or build further upon it. We also discuss the ethical implications of our approach and how we addressed these based on the Menlo report. Finally, we walk over the limitations our methodology and measurements have.

## 6.1. Security Implications of Orphaned Web Pages

As was demonstrated by the previous chapter, orphaned web pages indeed tend to be more vulnerable to security exploits compared to non-orphaned web pages. This shows that detecting orphaned web pages can be of great benefit to website administrators, therefore showing the need for our methodology. Because we are the first study to shed light on the concept of orphaned web pages, we briefly discuss how a vulnerability in these web pages compares to a vulnerability in non-orphaned web pages.

### 6.1.1. Similarities to Non-Orphaned Web Pages

Like many vulnerabilities, we suspect orphaned web pages to be no more than a security misconfiguration rather than a fundamental flaw in a protocol or service. From a technical level, exploiting an orphaned web page is no different than exploiting a non-orphaned web page. That is, any technique or proof-of-concept that works on non-orphaned web pages (for example, to exploit a Cross-Site Scripting vulnerability or SQL Injection) will also work on orphaned web pages. We therefore claim no zero-day approach to compromising a web application or server using the found orphaned web pages.

Earlier studies have uncovered details on the human aspect of security misconfigurations [18, 37]. From these studies, we extract many parallels to orphaned web pages. For example, it was found that a lack of documentation, lack of responsibility, and unclear processes and procedures can lead to security misconfigurations. With orphaned web pages, a lack of documentation could signify an outdated sitemap, or no sitemap at all. This can cause developers and administrators to lose overview of which page links to which, but also what pages are still online to begin with. As a result, it is easy to forget about a certain page or link, which can lead to orphaned web pages upon website restructuring (see Chapter 2 for more details on how a page becomes orphaned). Second, a lack of responsibility can cause misunderstandings within the development team as to who controls which page. Without a defined demarcation of responsibility, it can become unclear who is required to move a specific page, which can cause the page not to be removed at all, thus becoming orphaned. Lastly, vagueness in processes and procedures can lead to unplanned manual changes of a web page, which again is prone to neglectful behavior, and can hence lead to an orphaned web page.

### 6.1.2. Differences with Non-Orphaned Web Pages

One of the key characteristics of orphaned web pages is that they have become forgotten and unmaintained. Apart from the fact that these characteristics make orphaned web pages more prone to security vulnerabilities (which we have shown in Chapter 5), they also make it much more difficult to account for their risks. We mean by this the following: When a company makes its risk analysis, it takes into account all possible vulnerabilities for its web application and makes a trade-off based on the company's security requirements and risk appetite. If a company aims to eliminate, for example, all forms of Cross-Site Scripting attacks on its website it will invest in all necessary protections for the pages in its domain. However, since orphaned web pages are forgotten resources, they cannot be taken into account when putting in these measures, which leaves out an open risk that is unwillingly taken by the company. We link this back to the matrix of Figure 1.1: what a company does not know about, it cannot protect. A company can therefore only take adequate measures to protect their orphaned web pages *once it has become aware of* these orphaned web pages. This again argues the need for the methodology developed in this thesis.

Second, because orphaned web pages generally remain unmaintained, having the rest of your website up-to-date and equipped with adequate security measures can give a false sense of protection. No matter how advanced or thorough the security mechanisms are of a web application, one forgotten orphaned web page has the potential to fully get a web server compromised if it contains outdated software with a CVE. As such, orphaned web pages have the potential to be the Achilles' heel of a web application. Detecting these pages is therefore crucial to avoid a false sense of security.

The above mentioned reasons make vulnerabilities caused by orphaned web pages fundamentally different when it comes to the impact of a website's security, even though exploiting these vulnerabilities is for a large part the same as with non-orphaned web pages.

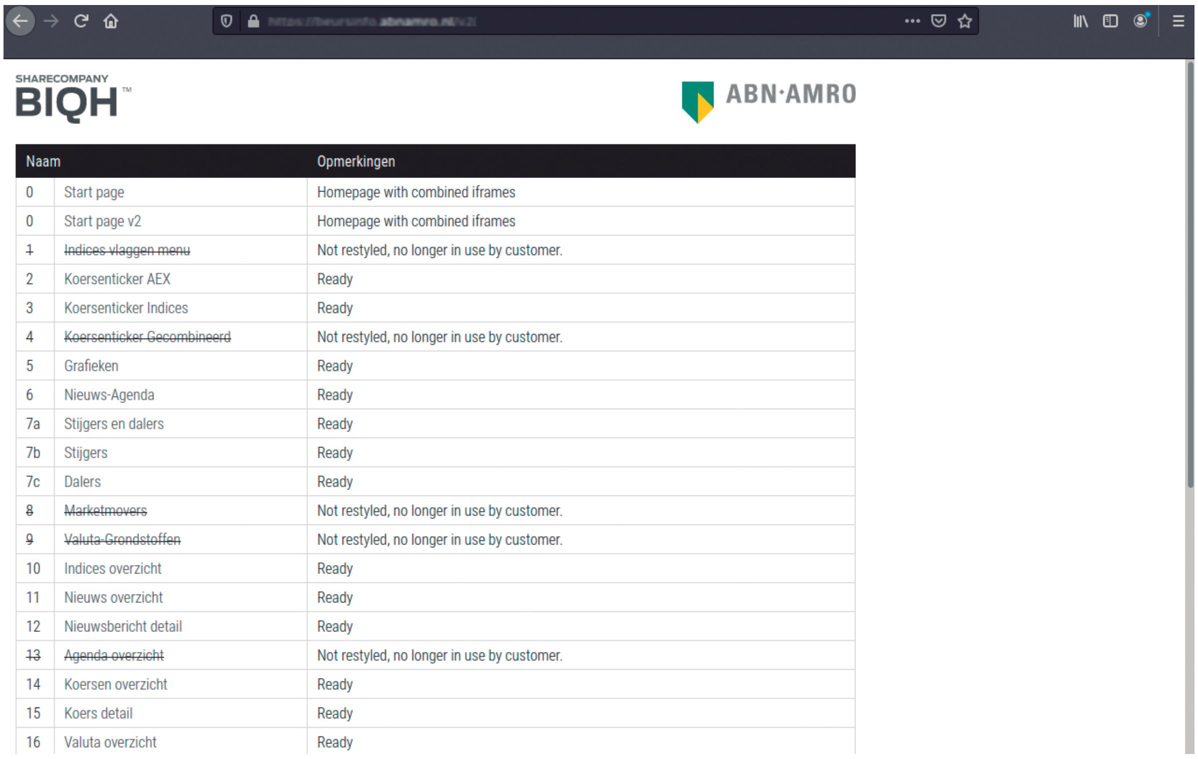
### 6.1.3. Collateral Security Implications

Although we have proven and discussed the direct security implications of orphaned web pages themselves, this is unfortunately not where an orphan page's impact on security stops. While orphaned web pages themselves were once linked to by a different page on the domain, they can also form a sort of 'gateway' toward other orphaned web pages or subdomains that were never linked to by any other web page in the website.

For example, let's assume a website hosted on the domain `example.com`. At some point in time, `example.com` used to have a link to the web page of `sensitive.example.com/development/test/page1.html` which is a simple boilerplate page functioning as a test to confirm successful deployment. The page contains no further content or code, and is hence not vulnerable to any security exploit. At some point, development for the subdomain `sensitive.example.com` is ceased, and the hyperlink to `sensitive.example.com/development/test/page1.html` is deleted from the website. Ergo, the web page has become orphaned. Using our methodology, we can detect the orphaned web page, although to not much avail since it contains no content or code. However, the subdomain itself (`sensitive.example.com`) might contain interesting data, as it is equally outdated and unmaintained as the initial orphaned web page. No other page on the subdomain should, however, be considered 'orphaned' since no previous ancestry link was present in the sitemap of `example.com`.

During our analysis, we found several of such pages. This happened mostly by removing part of the URL of a found orphaned web page until a web application was hit. Figures 6.1 and 6.2 show each an example of such pages. In Figure 6.1 we see what seems to be a progress overview of the development for a fiscal web application used by one of the main banks in the Netherlands. When we click the name of a page, we navigate to the (still under development) web page, which is online and connected to several other services such as, among others, stock market APIs. A live web application like this, still in development, was probably not meant to be discovered by the public (see also Figure 1.1). Yet, due to an exposed orphaned web page, it became trivial for adversaries to come across this subdomain. If security measures are not (yet) implemented in the developed web application, adversaries might be able to exploit it before the subdomain is 'officially' deployed. For ethical reasons, we did not attempt any attack on the web application (see Section 6.4 for our discussion on the ethics of this study).

As a second example, we found a login panel to what seems to be a cabin crew roster portal of a Dutch airline company. Presumably, the login panel is meant for internal use, and may therefore never be intended to be exposed to the public. The code of the web application (which is linked to via the source code of the web page) is also hosted on the subdomain an can be read by the user. However, as with the previous example, we never abused or exploited information/code encountered on the page, and we will elaborate more on the ethics of this study in Section 6.4.



| Naam | | Opmerkingen |
|---|---|---|
| 0 | Start page | Homepage with combined iframes |
| 0 | Start page v2 | Homepage with combined iframes |
| ~~1~~ | ~~Indices vlaggen menu~~ | Not restyled, no longer in use by customer. |
| 2 | Koersenticker AEX | Ready |
| 3 | Koersenticker Indices | Ready |
| 4 | ~~Koersenticker Gecombineerd~~ | Not restyled, no longer in use by customer. |
| 5 | Grafieken | Ready |
| 6 | Nieuws-Agenda | Ready |
| 7a | Stijgers en dalers | Ready |
| 7b | Stijgers | Ready |
| 7c | Dalers | Ready |
| ~~8~~ | ~~Marketmovers~~ | Not restyled, no longer in use by customer. |
| ~~9~~ | ~~Valuta-Grondstoffen~~ | Not restyled, no longer in use by customer. |
| 10 | Indices overzicht | Ready |
| 11 | Nieuws overzicht | Ready |
| 12 | Nieuwsbericht detail | Ready |
| ~~13~~ | ~~Agenda overzicht~~ | Not restyled, no longer in use by customer. |
| 14 | Koersen overzicht | Ready |
| 15 | Koers detail | Ready |
| 16 | Valuta overzicht | Ready |

Figure 6.1: Example of a (potentially sensitive) subdomain found through an orphaned web page. The page appears to display the progress of the development of a financial web application for a Dutch bank.

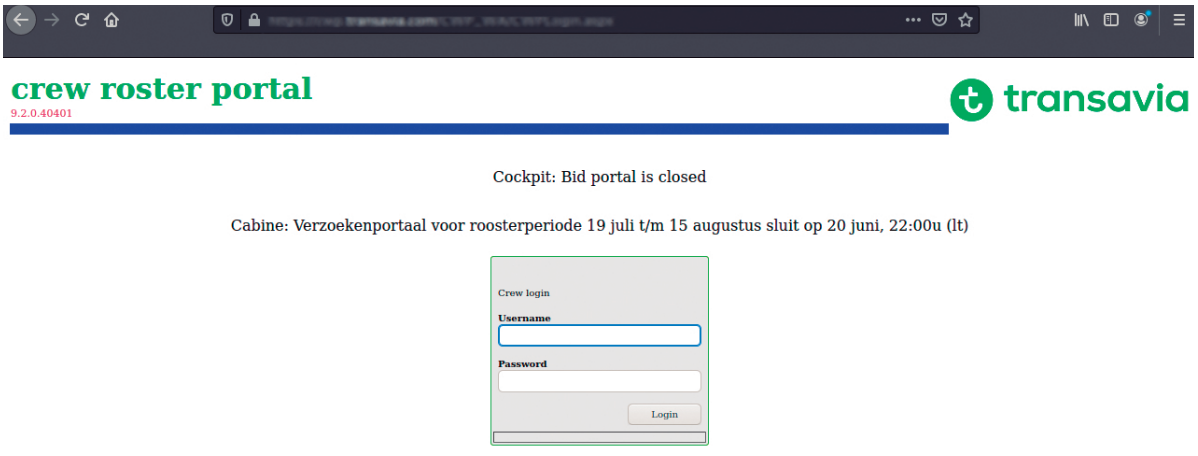
Coming back to our orphan page on `example.com`. It might be that, although `sensitive.example.com/development/test/page1.html` is not vulnerable to any security exploit, other pages on `sensitive.example.com`, such as the index page, do contain sensitive data, or are even vulnerable to an (outdated) security exploit. These pages would have initially been hidden from the public (quadrants 3 and 4 in Figure 1.1) since they were never directly linked to from the original domain. However, because a non-sensitive page *was* initially linked to by the original domain, it is much easier for adversaries to find the sensitive data on `sensitive.example.com` since there is an orphan page on that subdomain. This means that orphaned web pages can function as a stepping-stone toward outdated subdomains or web pages, which can potentially contain security vulnerabilities.

**Answering the Research Question**

(**RQ2.4**): *How does a vulnerability on an orphan page differ from a vulnerability on a non-orphan page when it comes to the security impact it can have on a website?*
When an orphaned web page gets exploited, it adds an extra element of surprise to the impact on a company due to its forgotten status. As such, unmaintained orphan pages are by definition not included in risk analysis models, making their impact potentially bigger. Moreover, orphan web pages have the potential to reveal other web pages or subdomains that are not (or no longer) meant for the public, therefore functioning as a stepping-stone toward potentially vulnerable pages.

## 6.2. Impact on Internet Debris and Measurement Pollution

We have shown earlier in this thesis that websites tend to increase in number of pages with each year (see Section 5.1). Later on became clear that this increase is in part unintentional due to some of these pages being orphaned, meaning they should have been removed at some point earlier in time. The result of this is that many pages somewhat keep ‘floating around’ online almost like a satellite in space whose mission is discontinued, yet remains floating in orbit around, for example, Earth. This is more commonly known as *space debris*. We can make the same analogy with orphan pages on the web contributing to some sort of ‘Internet


[redacted]

crew roster portal
9.2.0.40401

transavia

Cockpit: Bid portal is closed

Cabine: Verzoekenportaal voor roosterperiode 19 juli t/m 15 augustus sluit op 20 juni, 22:00u (lt)

Crew login
Username
Password
Login

Figure 6.2: Example of a (potentially sensitive) subdomain found through an orphaned web page. The page appears to be a login panel for the cabin crew roster portal of a Dutch airline company.

debris'. The more Internet debris we create, the more careful we need to be when doing Internet measurement studies. These orphan pages can namely give a distorted view, depending on what is measured. For example, measuring update behavior of system administrators by means of probing a web page at different points in time can make it seem as if administrators don't enforce updates if orphaned web pages are involved in the study. We argue that this is a false positive, since the administrator is most likely unaware of the page and, hence, updates the rest of the website more properly. Luckily, we have shown that orphan pages are in general difficult to find without a tailored method, so we expect their impact on measurements to be very little. Nonetheless, they should be taken into account and, more importantly, should be actively searched for by website administrators to avoid the problem to escalate since it could lead to measurement pollution otherwise.

## 6.3. Applications

As mentioned before, we publish our methodology as an open source implementation. This, together with the modular design of our implementation, allows users from different areas to experiment with, and enhance, our methodology. The methodology we present in this paper can be used by website administrators and defenders (blue team), as well as security professionals (red team) to discover (potentially vulnerable) orphaned web pages. Furthermore, our work forms the foundation for a deeper exploration of orphaned resources on the Internet. For these specific three target groups (website administrators, security professionals, and researchers) we discuss the possible applications our implementation can have on their field.

### 6.3.1. Use by Network Defenders

In Chapter 2, we have shown various scenarios in which a web page can become orphaned. In those scenarios, the orphaning of a page slips by unnoticed, which is why it can lead to the web page becoming unmaintained, and thereby a security risk. Our implementation can assist defenders by automating discovery for outdated pages and web resources in general.

There are many ways in which a website administrator can tweak our methodology toward the needs of their analysis. Instead of (solely) relying on the Internet Archive as a baseline, the administrator can also make use of internally stored sitemaps of the website. These have the advantage that they can be more accurate and more detailed than the information from the Internet Archive, also allowing to scan pages excluded from the web crawls.

Furthermore, the use of Dynamic URL Detection (DUDe) can be configured differently for network defenders. For our measurement study, we specifically configured DUDe to realize the highest reduction percentage, therefore aiming to capture a lower bound on the prevalence of orphaned web pages. This unavoidably lead to true positives being removed from our data set for the trade-off of scalability. When using our methodology on just a single domain, however, it might be more beneficial to have a less strict configuration of DUDe, therefore capturing more potential orphan pages. In our implementation, this can be easily done by tweaking the popularity cutoff, short-link threshold, long-link threshold, and/or long-link cutoff. Moreover, since our implementation is modular, the DUDe module can be omitted completely from the pipeline, if desired.

Additionally, instead of only looking at orphaned web pages with an HTTP OK (200) status code, other responses might be of interest, such as status code 50x responses indicating applications that still exist, but, for example, lost access to their backend database. Similarly, the filtering of file extensions can be adjusted, or completely omitted, allowing the detection of orphaned resources like exposed documents.

We do expect, however, that website administrators need less processing steps post-probing due to their familiarity with the set of URLs. In our methodology, we try to filter out as much false positives as possible by making use of different heuristics. The administrator of a website, on the other hand, might recognize these false positives much quicker, and much more accurately, using manual inspection of the set of URLs. Where we would, for example, apply DUDe to guess dynamically generated URLs, a website administrator can instantly filter out all URLs on a specific path, using regular expressions, of which he/she knows for sure that they are not orphaned. The modularity of our tool supports this type of approach.

### 6.3.2. Use by Red Teams

Security professionals, such as pentesters, can also benefit from using our implementation in their arsenal. Common tools like DirBuster [59] or Gobuster [60] can detect (unintentionally) accessible pages on a domain, but they rely on the use of wordlists and brute-force enumeration. While this method is effective for finding conventional pages, like a login panel or upload platforms, our technique is complementary in that it provides a more targeted-search tailored toward the domain under investigation. It identifies specific end points potentially not included in common word-lists and provides an indication of pages that are not maintained anymore, meaning they are likely more vulnerable. Also Google Dorks [58] are commonly used for identifying orphaned or hidden sites. However, as we have shown in Section 5.7, our approach largely covers pages *not* indexed by Google. Similarly as with network defenders, security professionals can configure our implementation toward their analysis needs by modifying modules such as DUDe, file extension filtering, status code filtering, etc. Moreover, red teams can configure our implementation to stop before the step of filtering out file extensions. At that stage, our methodology merely outputs a list of pages that were once archived by the Internet Archive, but have no recently crawled version (ergo, these are potential orphan candidates). Since no file extensions are filtered out, red teams could use this list in combination with specific queries (through, for example, regular expressions) to find vulnerable files. This is similar to Google Dorks (where an adversary would look for sensitive files indexed by Google), only applied to potential orphan candidates. In fact, it might be interesting to query for sensitive files (i.e. `passwords.pdf, credentials.xls, users.txt, etc`) at any phase of our methodology, to look for confidential information that was once (accidentally) advertised online, yet removed at a later stage. We elaborate more on this in the future work section of Chapter 8.

### 6.3.3. Use by Research Community

Our technique provides the first baseline for studying the prevalence of orphaned web pages on the web. Further research should refine our technique to also provide an upper bound of orphaned resources. We further elaborate on future work topics in Chapter 8.

Apart from providing the first methodology on detecting orphaned web pages in the wild and at-scale, we also describe a nomenclature and general definition of this phenomenon (see Chapter 2), which acts as the basis for future research studying this topic. Having exact definitions and consistent jargon helps to keep research on this topic streamlined and coherent, which is essential when a new concept is introduced to the body of literature.

By making our technique open source, it is not only possible to build further upon our results, but also to reproduce them. Along with the code of this project, our repository[1] contains the random sample used from the Tranco Top 1M, yielding full transparency into the domains used for our study.

## 6.4. Ethics

Our work uses active measurements. As such, we follow established best practices as outlined in the Menlo report [5, 19]. We took precautions to limit the operational impact of our work on the web archive in coordination with the operators, and on the measured sites by limiting and randomizing the number of requests a site receives (see Section 4.3).

We included contact information in the User-Agent of all requests that we made for this project. Two parties opted-out of our research, and we excluded them from it.

Our HREC (Human Research-Subject Ethics Council) does currently not accept applications for ethical evaluations or waivers for research that does not directly involve human subjects. Hence, for the automated security scans we carefully considered the trade-off between the utility of our research vs. the potential harm we cause. As we made sure to only *detect* vulnerabilities and did not exploit them, we consider our approach ethical. Naturally, we disclosed all vulnerabilities to the administrators of the websites containing an orphan page with a XSS or SQLi vulnerability, or to the respective CERT when no direct contact information was available.

## 6.5. Limitations

Performing large-scale measurements inevitably has its shortcomings, to which our study is no exception. It is therefore important to note these limitations for correctly assessing the impact of our results. The following will discuss the limitations of specific aspects from this study and how they may effect the results. Later, in Chapter 8, we reach back to some of these limitations as to how they can provide inspiration for future work.

### 6.5.1. Implementation Details

During the design process of our orphan detection methodology, several trade-offs and decisions had to be made, which may impact our results. The first is the filtering of all HTTP responses, except for OK (200) (Section 4.3.1). While it guarantees our list to contain only active and responsive web pages, it does potentially miss out on other orphaned web pages returning a different status code than OK (200). If other status codes would be included, however, an additional module would need to be added to the toolchain which allows to interpret the different status codes to eventually receive a valid page (such as, for example, following a 301 Redirect response). This might significantly slow down the toolchain, making it perhaps less fit for large-scale measurements.

Second, the initial filter we apply for identifying custom error pages at-scale (that is, pages with an error message that nonetheless return a status code of OK (200)) also has the potential for missing out on true positives (Section 4.3.2). The heuristic works by checking the size of each web page, and deleting all entries from the same domain that have a size within 5 bytes of difference. This can early-on filter out valid orphans, such as boilerplate pages or login panels. Although we do a more refined check for these pages later on in the pipeline, doing this at an earlier stage can result in detecting more orphan pages, be it at the price of less speed and, hence, scalability.

Lastly, the methods used for detecting copyrights, following redirects, and loading frames are built for the purpose of a rudimentary check and are, hence, not complete (Section 5.4). We do not cover every possible in-code redirect, and our methodology for loading frames focuses on the first encountered web page. Developing more ways of following redirects and loading frames can provide additional depth to our page type analysis. Similarly, detecting more copyright statements (for example, different copyright standards, or in-code construction) can improve accuracy. Especially the detection of *dynamic* copyright statements can contribute to expanding our results.

### 6.5.2. On the Concept of Lower Bound

A central theme throughout this thesis is that of wanting to establish a *lower bound* for orphaned web pages. We believe it is important to state a lower bound to prevent an alarmist response and alert fatigue. Our implementation provides an accessible aid for users to assess the severity of this problem on their domain, while simultaneously inviting further research. This means that all heuristics employed in our study (Dynamic URL

[1] Available at: `https://github.com/OrphanDetection/orphan-detection`.

Detection heuristic and fingerprint comparison) are configured conservatively to reduce the chance of over-matching. However, it also inevitably causes us to miss potentially orphaned web pages, meaning orphaned web resources are likely more prevalent.

### 6.5.3. Reactionary Policies to Measurement Probes

Receiving measurement probes on one's domain might not always be the desired type of traffic for a website administrator. This can have many reasons, of which the most straightforward one is the unwanted increase in server load for the domain. When probing our list of potential orphan candidates, we provide a customized User-Agent that allows targets to contact us and potentially opt-out of our study. As mentioned before, two parties followed up on that invitation, which resulted in their domains being excluded from further probing. We cannot predict, however, how many administrators took different measures to handle our presumably unwanted probes. For example, after receiving $x$-amount of probes, a server might block any incoming traffic from the IP addresses stemming from our measurement servers. Since the response of the pages on this server will not be a status code OK (200), these pages will be filtered out of our study. Although we put measures in place to randomize the probes, and hence spread out the probes per domain over time, we cannot guarantee that our IPs would not be blocked, or that a different status code will be returned after a certain amount of probes. The effect again would be that orphaned resources could likely be more prevalent.

### 6.5.4. Data Set Bias and Completeness

Finally, the data sets used for this study also contain some bias. Concretely, we used data from the Tranco Top 1M [33] and Internet Archive [3]. From the Tranco list, we took a sample of 100k domains stemming from the top 500k entries. Since the list is a combination of multiple Top 1M lists, it has a bias for popular domains, specifically the 1 million most popular ones. The prevalence of orphaned pages may therefore differ for less prominent domains. Furthermore, the Internet Archive may be incomplete due to the limitations of their own crawler implementation. The Internet Archive also depends on 'crawl donations', which is archive data stemming from external crawlers. These crawlers inevitably come with their own limitations. However, we believe that these limitations have no significant impact on our result, as the extraction of archive data is one of the initial steps in our pipeline and is, hence, followed by many filters and heuristics.

# 7 Related Work

Creating scientific output gives an imperative feeling of standing on the shoulder of giants. This work, too, stands on the metaphorical shoulders of many giants, all in the form of previously published scientific work. We therefore make an overview in this chapter of the related literature that lays at the basis of this study. Specifically, we identify two main areas of research that are closely linked to our study: 1) prior work related to orphaned resources on the Internet (not limited to the web) and use-after-free attacks on these, and 2) literature on update behavior and measuring security misconfigurations in general.

We start with discussing the literature from both areas, in which we dive deeper into some earlier results and approaches from similar studies as the one in this thesis. Followed by this, we highlight the research gap currently existing that this work helps to fill.

## 7.1. Orphaned Resources

We explained and formerly defined the concept of orphaned web pages in Chapter 2. However, the concept of an 'orphaned resource' can go much broader than just merely web pages on a domain. This is due to its natural evolution out of the use-after-free concept in (low-level) programming languages such as C. In C, pointers are used to references addresses of memory locations where program information (such as variables) are stored. If a pointer no longer has a chunk of memory allocated, yet is being used in the program, memory safety violations will occur potentially leading to security vulnerabilities. CVE-2014-1776 is an example of such a 'use-after-free' vulnerability in Microsoft Internet Explorer 6 through 11, which enables adversaries to execute arbitrary code or potentially cause a Denial of Service (DoS) [41].

In 2010, Kalafut et al. studied the prevalence of orphaned DNS servers on the Internet [32]. They define orphaned DNS servers as those that do have a DNS record pointing to them, while they do not receive any delegation from a parent zone. They analyze zone files by retrieving all `A` records. For every found domain, the zone file is again used for its parent domain, i.e. if `ns1.example.com` exists the algorithm will look for `example.com`. If no results are found, `ns1.example.com` is labeled as orphaned. In their study, Kalafut et al. found that 1.7% of the records in their study's zone file belong to orphaned DNS servers. However, they also find that the majority of these orphans have a rather short lifespan, with 12% lasting less than a single day. Interestingly, a very small fraction of these servers are used for malicious purposes, like scam-websites and spam.

A decade later, Sommese et al. did a follow up study on this to look whether the DNS landscape made any adjustments to mitigate orphaned misconfigurations [48]. The authors concluded that, unfortunately, `.com` and `.net` were the only TLDs able to reduce the amount of orphaned records to zero. Moreover, Sommese et al. conclude that some TLDs experienced an increase in the number of orphaned records, showing that the original problem introduced by Kalafut et al. got worse.

Liu et al. [38] demonstrate how through 'Dangling DNS Records' (called Dares in the paper) domains can be hijacked by adversaries. A Dare is defined as an existing DNS record pointing to a resource that is no longer valid. This can be hijacked by adversaries who can then use it for malicious purposes. To study the prevalence of Dares, Liu et al. performed a large-scale measurement study and found 467 Dares that can be exploited. They identify four vulnerable types of DNS records, namely `A`, `CNAME`, `MX`, and `NS`, that can be exploited through abandoned third-party services, expired domains, or cloud IPs.

The latter attack vector has been studied more extensively by Borgolte et al. [10]. More specifically, the authors investigate IP address use-after-free vulnerabilities as the basis of domain takeover attacks. Using large-scale measurement studies, the authors found stale DNS records pointing to cloud IP addresses. Once these cloud IP addresses become free, and adversary can hijack the address and take profit of the still existing DNS record. Furthermore, Borgolte et al. also show that these attacks are practical to execute and effective on public clouds, primarily due to these freed IP addresses which are still the target of `A` records.

The vulnerability of stale `NS` record types has been further studied by Alowaisheq et al. [2]. The authors introduce the "Zombie awakening attack," in which stale `NS` records are being exploited through DNS hosting providers, again leading to domain hijacking. An `NS` record becomes stale when a costumer stops using a hosting service, but forgets to clear up the remaining `NS` records pointing to the hosting service. Because hosting providers do not validate domain ownership, an adversary is able to claim the domain of the customer at the DNS hosting provider. After a process of cache poisoning, the attacker eventually is able to divert lookups towards her malicious IP rather than the correct IP of the customer. Through measurement studies, Alowaisheq et al. identified 628 domains (out of the Alexa's 1M data set) that could potentially be hijacked.

Beyond DNS, Gruss et al. [25] show that the concept of use-after-free also applies to expired free-mail addresses. Free-mail addresses are email addresses originating from free-mail providers, which offer free email for people who do not have their own mail server. A use-after-free attack on these free-mail addresses becomes possible once an account is deleted, or expired automatically after a given period of inactivity. Gruss et al. confirm that use-after-free attacks can cause exposure of sensitive information and account access when applied to free-mail services. Additional analysis shows that 33.5% of leaked email addresses in their study are indeed in an expired state, hence, making them vulnerable to use-after-free attacks.

Related to the web, Aturban et al. examine the existence of orphaned annotations in Hypothes.is, a tool for making notes and annotations to websites [4]. An annotation is considered orphaned when it can no longer be attached to its target web page, either because the web page does not exist anymore, or because the annotated content on the web page has been removed. In their study, the authors found that 27% of text annotations can no longer be attached to a live version of the target website. However, 3% of these annotations could be reattached to archived versions of the target website by making use of web archives. This indicates the potential of using web archives to obtain orphaned data; a concept we also leverage in this work.

Similarly, research done by Harvard in collaboration with the New York Times shows that many old hyperlinks, referenced to by news articles, were not accessible anymore [66]. This is commonly known as *linkrot*. The authors analyzed over 500,000 articles stemming from `nytimes.com` between 1996 and 2019. Over 2 million hyperlinks within those articles point to a website external to `nytimes.com`, with 72% of the URLs referencing a specific path on the domain (called "deep links"). From those deep links, 25% were not accessible anymore, showing that linkrot is indeed very much present. Interestingly, the authors confirm that the older articles (and, hence, older web pages) are more prone to containing a rotten link. We show a similar result with respect to orphaned web pages, noticing an equal trend between amount of orphan pages and the age of a page. Also like our study, the authors make use of the Internet Archive to study this phenomenon, showing again the potential of web archives to study old (forgotten) web pages.

### 7.1.1. Orphaned Resources vs. Use-After-Free

Orphaned resources in the related work focus on resources that are no longer allocated themselves, but still receive a form of delegation from another service. Terminology for this situation usually differs—'stale', 'dangling', 'use-after-free', and sometimes 'orphaned'—while the underlying concept remains the same. In contrast to these broader concepts, we focus on resources and services that lost their delegation, while continuing to exist, i.e., effectively the inverse type of orphaned resources in comparison to the literature. In our work, the resources in question (web pages) cannot be re-used or hijacked when an attacker takes over an orphaned *delegation*. Instead, the resource itself becomes the liability, e.g., due to aged software and unpatched vulnerabilities. Nevertheless, orphaned web pages are still under the original author's control. Other terms, such as use-after-free, signify that control has been relinquished and the resource can be taken over for exploitation.

## 7.2. Update Behavior and Misconfigurations

The second area related to our study is that of security misconfigurations. While some of the previously mentioned studies investigate misconfigurations, we separately discuss the relevant studies done on update behavior and security misconfigurations *outside* the realm of orphaned resources.

Orphaned web pages are a form of security misconfigurations. Dietrich et al. studied the perspective of system administrators on security misconfigurations, providing insight into the human aspect of this phenomenon [18]. By means of surveys, the authors found that causes for human error often reside in institutional, organizational, and personal factors. We see intuitive parallels between the issues uncovered by the study and potential causes of unidentified orphaned pages, such as a lack of documentation (non-updated sitemaps), lack of responsibility (who controls which page), and unclear processes and procedures (unplanned manual change of pages). Although we would like to stress that none of these causes were verified by our study and are, hence, mere speculation based on the literature.

The work of Li et al. also suggests that orphaned web pages might be an issue of misconfiguration [37]. The authors focus specifically on the updating *practices* of system administrators, and report on two relevant causes –found through surveys– of outdated and vulnerable software, namely deployment within a timely manner, and adequate testing. These two factors can easily be applied to web application development, therefore suggesting that similar causes might be present for orphaned web pages. While we did not interview or survey any of the web administrators involved with the detected orphan pages, literature suggests that forgotten or unmaintained web pages can indeed be expected to manifest into the wild as a cause of human error (rather than, for example, technical failure). Albeit we confirmed their prevalence, future work is needed to confirm whether its origin lays within human error (See Chapter 8).

Despite survey studies, the topic of security misconfigurations holds a rich body of measurement studies on the prevalence of specific security misconfigurations, especially focusing on at-scale measurements. While survey-based studies can provide insight into the *causes* and *perspective* of security misconfigurations, measurement studies are essential to put them into context and capture their prevalence, as well as impact, in the wild.

For example, Continella et al. built a tool to detect misconfigurations around Amazon S3 cloud storage services [16]. Similar to our study, they perform large-scale measurements to both verify the efficacy of their method, as well as capture the prevalence of these misconfigurations in the wild. They find that 14% of their scanned S3 buckets were public, with 2% offering write access. Furthermore, Continella et al. show how particular buckets connected to web applications can deface a website, or even deliver malicious content to the end users.

A similar study was done by Ferrari et al. on discovering misconfigured NoSQL services [20]. Although most NoSQL services come with built-in security features, they are not often enabled by default. Moreover, such security features are often cumbersome to configure, especially for non-expert users. The authors therefore did a large-scale measurement study to detect exposed NoSQL services, and identify unprotected instances. Analyzing a little less than 68 million IP addresses, Ferarri et al. found 12,276 vulnerable instances. They furthermore demonstrate two main security issues with these instances: 1) the exposure of sensitive private information such as email addresses, usernames, and passwords, and 2) more than 1,700 reachable websites were linked to these exposed NoSQL instances, of which 28.6% is potentially vulnerable to defacement, and 17.3% to arbitrary code injection.

Earlier, Springall et al. measured the prevalence of misconfigured FTP servers [49]. The File Transfer Protocol (FTP) is, as the name suggests, a protocol for transferring files, allowing them to be shared among hosts, and making them available for widespread distribution. FTP services have the option to allow anonymous access, which makes it feasible for the public to retrieve data form the FTP server. In most cases, this can be considered as a misconfiguration, as sensitive data is unwillingly exposed through these services. In their study, Springall et al. captured the prevalence of exposed FTP services by means of Internet-wide scanning. They show that, out of 13.8M FTP servers, 8% allowed for anonymous access.

The work of Bijmans et al. studies a vulnerability in MikroTik routers which allows adversaries to inject cryptomining code into user traffic [8]. Concretely, the authors found that, due to this vulnerability, all traffic of any user can be infected by cryptomining code, essentially 'hijacking' a computer to mine cryptocurreny with every website they visit, yielding profit to the adversary. This is better known as *cryptojacking*. Using large-scale measurement data, Bijmans et al. show how over 1 million MiktroTik routers are vulnerable for this attack. Furthermore, the authors shed light on some of the campaigns actively making use of this vulnerability.

Also DNS(SEC) misconfigurations has been widely proven to cause unavailability and security issues [61, 17, 28]. Van Adrichem et al. evaluate DNSSEC misconfigurations for domains in the .bg, .br, .co, and .se zones [61]. They find that 4% show indeed misconfigurations, and that 75% of them were unreachable from a resolver using DNSSEC. This is primarily due to inconsistencies in the DNSKEY. This violates the security of these domains, more precisely the availability of it.

Continuing on DNSSEC, Dai et al constructed a tool designed to collect and analyze signed domains for DNSEC misconfigurations [17]. Similar to our study, they evaluate their tool by the means of Internet-wide measurements. The authors report that a little less than 20% of the studied Alexa domains contain a cryptographic failure, therefore not allowing them to establish a chain of trust.

Another type DNS misconfigurations are the so called *lame delegations*. A lame delegation occurs when a nameserver cannot provide authoritative answers concerning a specific domain, even though it is a delegated authority for that domain. In a study by Akiwate et al. the prevalence of these delegations was estimated at around 14%, which was calculated using a combination of large-scale passive and active DNS measurements [1]. These lame delegations pose a security risk, as the nameserver domain could be registered by an adversaries, allowing her to hijack potentially thousands of domains.

Lastly, Hendriks et al. performed a study on misconfigurations concerning IPv6-specific DNS records [28]. With the rise of IPv6, we see more and more IPv6-only setups in networks, therefore not providing a fallback on IPv4 in case of misconfigurations and/or erroneous behavior. By making use of long-term active DNS measurements (spanning multiple zones), Hendriks et al. show that over 97% of invalid records are due to one of ten major misconfiguration types.

These studies underline how misconfigurations are still a frequent cause for security issues, and are regularly occurring in the wild.

## 7.3. Research Gap

We have seen from the related work that orphaned resources can appear in multiple areas of computer systems. To no surprise, DNS is by far more susceptible to orphaned resources [32, 38, 2], but we have also seen work on orphaned cloud IPs [10] (although used with DNS) or free-mail addresses [25]. In this study, however, we are the first to investigate the possibility of orphaned resources in *web applications*. As was explained in Chapter 2, the term 'orphaned' signifies the parallel of a page in a sitemap (child node) no longer being linked to from any other page in the sitemap (parent nodes), yet remains online and hence directly accessible through its URL (should that URL be found). As such, we are the first to provide a methodology for finding orphaned web pages in the wild, and proved that our technique is scalable by using it in a large-scale measurement study.

What was imperative in all of the studied related work was the need of resource hijacking to successfully use the orphaned resource as an attack vector. This is an essential difference compared to our work. In this study, the resources in question (web pages) are not intended to be re-used or hijacked after their omission within the sitemap, but it is rather their susceptibility to aged software and un-patched vulnerabilities that make them of interest to this study. Other terms, such as use-after-free, signify the requirement of takeover in order to successfully exploit the concept, which is not at hand in the phenomenon we describe.

## 7.4. Research Similarities: Lessons From the Related Work

Apart from defining a research gap, studying the related work can also help to identify best-practices and techniques that can be applied to our research. We therefore briefly highlight our similarities with the related work, and show how we use these to study the research gap mentioned earlier.

The main similarity among almost all studied works is the use of large-scale measurements. They are mainly used to show the prevalence and security impact of a vulnerability in the wild. Measuring such aspects allows to put the theory of an attack into context and asses its severity in contrast to other attacks. Furthermore, measurement studies can give insights into the feasibility and practicality of attacks, especially on aspects such as scalabity, speed, memory consumption, and financial requirements. Using large-scale measurements for this study is therefore crucial in answering the research question and, hence, filling the research gap.

Another peculiarity that stands out from the literature is that orphaned resources are almost always paired to a security consequence. Apart from the studies by Aturban et al. [4] and Zittrain et al [66], all papers from the related work report on a security vulnerability as a consequence of the orphaned resource under scrutiny. This motivated us to, apart from studying the prevalence, also study the security status of orphaned web pages in the wild. The assessment of the security impact, however, is always different in each study and depends on the orphaned resource. While most opt for the approach of creating a proof of concept for the attack, we showed a wider range of attack vectors by performing rudimentary security scans. What is similar, however, to all studies is the careful attention to the ethical considerations of assessing a resource's security, something we also put great importance on when performing the security scans, as well as the large-scale measurements (see our ethics discussion in Section 6.4).

# 8 Conclusion and Future Work

In this thesis, we introduced the concept of orphaned web pages, and built the first methodology to systematically detect these pages in the wild. Apart from providing a lower bound on the prevalence of orphaned web pages, we also scrutinized the security posture of these pages, and discovered they are much more prone to (common) vulnerabilities compared to non-orphaned web pages. Based on the results of our study, we can adequately answer the research question posed in Chapter 1. We do so in an incremental fashion, by splitting the main research question up into sub-questions, and linking back each of our contributions to these sub-questions. Due to the novelty of our work, new research opportunities naturally emerge from our findings, which we briefly discuss in this chapter as well. To conclude this chapter, as well as the thesis in its whole, we give a brief summary of our work, along with the impact it brings to the community.

## 8.1. Answering the Research Question

To recapitulate, the objective of this work is to provide the first comprehensive analysis of orphaned web pages in the wild and at Internet-scale. As such, the main research question for this thesis goes as follows:

> **(RQ):** What is the prevalence and security impact of orphaned web pages?

This question is further split into two parts: the prevalence and the security impact. We answer both individually.

### 8.1.1. Prevalence of Orphaned Web Pages

Studying the prevalence of any phenomena on the Internet can almost never escape the necessity of a large-scale measurement study (see Chapter 7). Ours being no exception, we required a thorough definition and understanding of the nature of orphaned web pages to design a suitable methodology for detecting these pages in the wild. Our definition and background of orphaned web pages can be found in Chapter 2, and our methodology based on this definition and background is described in Chapter 3. Using this methodology on a subset of domains stemming from the Tranco Top 1M list [33], we deployed our methodology to perform a month-long Internet-wide measurement aimed at detecting orphaned web pages in the wild. The results of these measurements (as described in Chapter 4.1 and Chapter 5) concluded that there are at least 1,953 orphaned web pages, spread across 907 domains in our data set, with some of these pages being as old as 20 years. This lower bound shows that orphaned web pages are in fact present in the wild, and that they can easily be detected by our methodology.

### 8.1.2. Security Impact of Orphaned Web Pages

Having gathered a set of orphaned web pages, we were able to inspect them more thoroughly and assess their susceptibility to security exploits. Doing so, we required a control group of non-orphaned pages to compare our findings against, since comparing against results from previous work will always have shortcomings and inaccuracies due to differences in test setups. Using Wapiti [54], we performed a black-box security scan on both groups, which was based on fuzzing techniques and checks for best practices. We conclude that orphaned web pages are in fact more likely to contain a security issue or vulnerability, with the orphan group having around 7.1% of their pages vulnerable to Cross-Site Scripting, compared to the control group having only 0.9% ($p < 0.01$ using $\chi^2$). Also regarding minor security issues such as Secure HTTP Headers or Content Security Policy, we notice that orphaned pages perform significantly worse compared to our control group. Furthermore, we have shown that the impact of a security vulnerability in an orphan page can potentially be more harmful due to its forgotten and unexpected nature, and that orphan web page can expose sensitive subdomains that were never (meant to be) public before. As such, orphan pages have the potential to be the Achilles' heel of a web application's security infrastructure.

## 8.2. Future Work

As much as we were inspired by concepts and techniques stemming from previous work, we hope that our study can achieve the same for research that is still to come. As such, we discuss possible future directions that can evolve from our result, and elaborate why they are interesting to explore.

### 8.2.1. Methodology Improvement

While our technique for detecting orphaned web pages is effective, we believe different approaches in detecting these pages are worth exploring.

**Historic Sources** We based our analysis on the available website data in the Internet Archive (IA) [3], and used it to compare sitemaps over time to find orphaned URLs. While the IA contained data on over 96% of the domains in our study, there are many other sources that could lead to potentially orphaned web pages. For example, blogs, fora, news websites, and social media platforms all keep a records of the historic content on their website. Scraping old pages from these sources (such as, for example, Twitter tweets) can potentially reveal old URLs pointing to orphaned web pages. Extracting these URLs from social media and/or other websites could be more robust against the limitations (crawl heuristics and depth) of web archives.

**Additional Orphaned Resources** As mentioned before in Section 6.3.2, also other orphaned resources, like .pdf or .xls, can be interesting to detect. Beyond orphaned web pages, we have seen Google Dorks being used to perform targeted searches for exposed files containing sensitive information such as passwords or addresses [58]. A similar approach can be used for orphaned web pages, where our technique is enhanced with efficient search queries (using, for example, regular expressions) to identify sensitive files that should not (or no longer) be online.

**Page Analysis** To classify the web pages, we analyzed them for copyright statements, boilerplate codes, error pages, redirects, and frames. Based on this classification, we could categorize the pages into likely orphaned, uncertain, and not orphaned. However, our techniques for doing so are rudimentary, and functioned mostly to find a lower bound, at-scale. Therefore, improvements can be made to refine our classification, potentially revealing more orphaned web pages. This can be, but is not limited to, detecting dynamically generated copyright statements, include more in-code redirect procedures and implement interpreters to reach their (potentially orphaned) redirected-to page, and developing a way to find the correct frame on a web page that contains the copyright statement.

### 8.2.2. Expanding the Lower Bound

The improvements mentioned above can help to expand the lower bound of this study, and, hence, reveal more orphaned web pages existing in the wild. A more general approach, however, to expand this lower bound is scaling up the measurement study. This can be done by 1) looking at more than 100,000 domains from a Top 1M list, and 2) including a fixed percentage of domains from all TLDs (i.e. 20% of .com + 20% of .org + 20% of .eu + ...), as has been done in other measurement studies [7].

### 8.2.3. Detecting Forgotten Domains

We can take the concept of an orphaned resource one level higher and look at forgotten, and potentially unmaintained, domains. Initially, we see two ways in how to detect such domains:

1. **Certificate Transparency (CT) logs:** To prevent fraudulent use of SSL certificates, CT logs maintain a public list of trusted certificates, which can be audited to monitor for inconsistencies [26]. Whenever a certificate is attributed to a domain, it can be pushed to the log, along with its meta data such as start date and expiration date. Although not all certificates are present in these logs, it has been shown

that these logs contain more than 90% of the certificates around [62]. If a certificate is expired and not renewed, it might be because the owner does not want to support SSL anymore, because the owner does not want to host the domain anymore, or because the owner simply forgot. In the latter two cases, if the owner forgets to take the domain offline, we might have a forgotten and unmaintained domain. By probing all domains in the CT log with an expired domain, and doing a similar analysis as our study, we can detect these domains and asses their security posture.

2. **Passive DNS:** The Domain Name System (DNS) is a system used to retrieve address information (such as an IPv4 address) of a domain name. This happens in a hierarchical way using dedicated DNS resolvers. If an entry is not cached by a resolver, the resolver will query external servers to eventually obtain the address information of a domain. Passive DNS data keeps track of these external queries, and keeps a log of which domain is associated with which IP address over time [44]. Such data can be used to depict the amount of traffic a domain receives over time [9]. Potential forgotten domains can be detected by looking at those domains who suddenly receive less traffic (indicating the domain advertised its termination to its users), yet remain online, potentially due to the owner forgetting to actually take down the domain.

### 8.2.4. Surveying Operator Perspectives

While our study was successful in detecting orphaned web pages, we got only very little insight into what the actual causes are. We portrayed some theory in Chapter 2 on how an orphaned web page can manifest from a technical perspective, and we could speculate based on the literature in Chapter 7 that the causes of these pages might be rooted in human error. However, we have no empirical evidence for this claim, which could be an interesting topic for future work. Concretely, website operators could be instructed to use our implementation for finding orphaned web pages on their domain. By surveying these operators, we could gain interesting insights into the prevalence (on a per-domain basis) and root-causes of orphaned resources. Recently, we have seen more survey studies being used to gain insight into the perspective of administrators and operators on security misconfigurations and issues [18, 37], which can serve as a basis for future work surveys.

### 8.2.5. Additional Security Evaluations

The security analysis done in this work is primarily focused on Cross-Site Scripting and SQL Injection, among some other (minor) security issues. Although sufficient for the purpose of this study, many other angles can be explored in analyzing the security posture of orphaned web pages. For example, rather than merely performing black-box fuzzing techniques, white-box approaches can reveal particular cases, for example through static code analysis, which cannot be discovered by black-box approaches. Additionally, it can be interesting to look at how orphaned web pages can be used as a stepping-stone toward other vulnerable pages or subdomains. We have briefly touched upon this in Section 6.1.3 by showing examples found through manual inspection. Capturing such pages/subdomains at-scale can therefore be an interesting topic for future work.

### 8.2.6. Orphan Web Pages and GDPR

Finding pages that are not intended to be publicly available (anymore) not only comes with potential security issues, but also with potential privacy issues. Apart from the obvious concerns, such as leaking password files or user lists, an orphaned web pages could also be a valid profile page that is incorrectly removed, therefore exposing personal information such as the address or phone number of an individual. This potentially violates privacy laws, such as the General Data Protection Regulation (GDPR) in Europe [65]. Moreover, we can extend this issue further to archive databases in general. Concretely, a web archive database might provide access to personal information which was asked to be removed from the original domain, regardless of whether or not the original domain indeed removed that data. If a web archive crawled the data before its removal, it is potentially reachable through the archival system. Whether or not this complies with the "Right to be forgotten" is an interesting case study for future work.

## 8.3. Final Remarks

The World Wide Web is indeed an ever-changing landscape. Keeping up with current-day standards is therefore no sinecure, making it often difficult to have up-to-date software and adequate (security) configurations.

Often rooted in human error, outdated software and security misconfigurations potentially lead to vulnerable systems. It is therefore crucial to be aware of these pitfalls, and avoid them at all cost.

In this thesis, we presented a new type of misconfiguration error, called *orphaned web pages*, and showed the potential security risks that comes along with them. Using large-scale measurements, we demonstrated that these pages are in fact prevalent in the wild, and that their security posture is much weaker than that of regular web pages. Concretely, orphaned web pages tend to be more vulnerable for trivial exploits such as Cross-Site Scripting, and can also cause unknown pages and subdomains to be exposed to the public, potentially with incomplete security measures or sensitive information. To bring more awareness to this problem and help website administrators to effectively detect orphaned web pages on their domain, we developed an open source implementation of our methodology that can detect these pages on a per-domain basis. We therefore hope to, with our work, make web administrators more aware about the existence of these pages, and the impact they may have on web application infrastructures. We also hope that our work, along with the open source implementation, furnishes more scientific research on this topic (see Future Work section above), as it was previously unstudied and therefore unknown to the scientific community.

# Bibliography

[1] Gautam Akiwate et al. “Unresolved Issues: Prevalence, Persistence, and Perils of Lame Delegations”. In: *Proceedings of the 20th Internet Measurement Conference (IMC)*. Ed. by Fabián E. Bustamante and Nick Feamster. Virtual: ACM, Oct. 2020. ISBN: 978-1-4503-8138-3. DOI: `10.1145/3419394.3423623`.

[2] Eihal Alowaisheq et al. “Zombie Awakening: Stealthy Hijacking of Active Domains through DNS Hosting Referral”. In: *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by Jay Ligatti and Xinming Ou. New York, NY, United States: ACM, Nov. 2020. ISBN: 978-1-4503-7089-9. DOI: `10.1145/3372297.3417864`.

[3] Internet Archive. *About the Internet Archvie*. URL: `https://archive.org/about/` (visited on 04/15/2021).

[4] Mohamed Aturban, Michael Nelson, and Michele Weigle. “Quantifying Orphaned Annotations in Hypothes.is”. In: *Proceedings of the 19th International Conference on Theory and Practice of Digital Libraries (TPDL)*. Ed. by Sarantos Kapidakis, Cezary Mazurek, and Marcin Werla. Poznań, Poland: Springer, Sept. 2015. ISBN: 978-3-319-24592-8. DOI: `10.1007/978-3-319-24592-8\_2`.

[5] Michael Bailey et al. “The Menlo Report”. In: *IEEE Security & Privacy* 10.2 (Mar. 2012), pp. 71–75. DOI: `10.1109/MSP.2012.52`.

[6] Sruthi Bandhakavi et al. “CANDID: preventing sql injection attacks using dynamic candidate evaluations”. In: *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson. Alexandria, VA, USA: ACM, Oct. 2007. ISBN: 978-1-59593-703-2. DOI: `10.1145/1315245.1315249`.

[7] Hugo L. J. Bijmans, Tim M. Booij, and Christian Doerr. “Inadvertently Making Cyber Criminals Rich: A Comprehensive Study of Cryptojacking Campaigns at Internet Scale”. In: *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. Ed. by Nadia Heninger and Patrick Traynor. Santa Clara, CA, USA: USENIX Association, Aug. 2019.

[8] Hugo L.J. Bijmans, Tim M. Booij, and Christian Doerr. “Just the Tip of the Iceberg: Internet-Scale Exploitation of Routers for Cryptojacking”. In: *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by XiaoFeng Wang and Jonathan Katz. London, United Kingdom: ACM, Nov. 2019. ISBN: 978-1-4503-6747-9. DOI: `10.1145/3319535.3354230`.

[9] Leyla Bilge et al. “EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis”. In: *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society (ISOC), Feb. 2011.

[10] Kevin Borgolte et al. “Cloud Strife: Mitigating the Security Risks of Domain-Validated Certificates”. In: *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. Ed. by Patrick Traynor and Alina Oprea. San Diego, CA, USA: Internet Society (ISOC), Feb. 2018. ISBN: 1891562-49-5. DOI: `10.14722/ndss.2018.23327`.

[11] Günter Born. *Deloitte-Seite ‘Test your Hacker IQ’ legt Benutzerdaten offen*. Nov. 6, 2020. URL: `https://www.borncity.com/blog/2020/11/06/deloitte-seite-test-your-hacker-iq-legt-benutzerdaten-offen/` (visited on 03/31/2021).

[12] Andrei Broder et al. “Syntactic Clustering of the Web”. In: *Comput. Networks* 29.8-13 (1997), pp. 1157–1166. DOI: `10.1016/S0169-7552(97)00031-7`.

[13] Jocelyn Chan. *The Different Types of XSS Explained With Code Examples*. Apr. 5, 2019. URL: `https://dzone.com/articles/the-different-types-of-xss-explained-with-code-exa` (visited on 06/05/2021).

[14] Moses Charikar. “Similarity estimation techniques from rounding algorithms”. In: *Proceedings of the 34th Symposium on Theory of Computing)*. Ed. by John Reif. Montréal, Québec, Canada: ACM, May 2002. ISBN: 978-1-581-13495-7. DOI: `10.1145/509907.509965`.

[15] World Wide Web Consortium. *HTML5 Differences from HTML4*. Dec. 9, 2014. URL: `https://www.w3.org/TR/html5-diff/#obsolete-elements` (visited on 06/04/2021).

[16] Andrea Continella et al. “There’s a Hole in that Bucket!: A Large-scale Analysis of Misconfigured S3 Buckets”. In: *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. Ed. by Juan Caballero and Guofei Gu. San Juan, PR, USA: ACM, Dec. 2018. ISBN: 978-1-4503-6569-7. DOI: `10.1145/3274694.3274736`.

[17] Tianxiang Dai, Haya Shulman, and Michael Waidner. “DNSSEC Misconfigurations in Popular Domains”. In: *Proceedings of the 15th International Conference on Cryptology and Network Security*. Ed. by Sara Foresti and Giuseppe Persiano. Milan, Italy: Springer, Nov. 2016. ISBN: 978-3-319-48965-0. DOI: `10.1007/978-3-319-48965-0_43`.

[18] Constanze Dietrich et al. “Investigating System Operators’ Perspective on Security Misconfigurations”. In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by Michael Backes and XiaoFeng Wang. Toronto, ON, Canada: ACM, Oct. 2018. ISBN: 978-1-4503-5693-0. DOI: `10.1145/3243734.3243794`.

[19] David Dittrich and Erin Kenneally. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research*. Tech. rep. U.S. Department of Homeland Security, Aug. 2012. URL: `https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf`.

[20] Dario Ferrari et al. “NoSQL Breakdown: A Large-scale Analysis of Misconfigured NoSQL Services”. In: *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*. Ed. by Kevin Butler. Austin, TX, USA: ACM, Dec. 2020. ISBN: 978-1-4503-8858-0. DOI: `10.1145/3427228.3427260`.

[21] The OWASP Foundation. *OWASP Top Ten*. URL: `https://owasp.org/www-project-top-ten/` (visited on 06/08/2021).

[22] Google. *Programmable Search Engine*. Nov. 10, 2020. URL: `https://developers.google.com/custom-search/docs/overview` (visited on 04/13/2021).

[23] Google. *Refine web searchers*. URL: `https://support.google.com/websearch/answer/2466433?hl=en` (visited on 06/05/2021).

[24] Google. *What is Programmable Search Engine*. URL: `https://support.google.com/programmable-search/answer/4513751?hl=en` (visited on 06/05/2021).

[25] Daniel Gruss et al. “Use-After-FreeMail: Generalizing the Use-After-Free Problem and Applying it to Email Services”. In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by Michael Backes and XiaoFeng Wang. Toronto, ON, Canada: ACM, Oct. 2018. ISBN: 978-1-4503-5693-0. DOI: `10.1145/3196494.3196514`.

[26] Josef Gustafsson et al. “A First Look at the CT Landscape: Certificate Transparency Logs in Practice”. In: *Proceedings of the 12th Passive and Active Measurement (PAM)*. Ed. by Mohamed Ali Kâafar, Steve Uhlig, and Johanna Amann. Vol. 10176. Lecture Notes in Computer Science. Sydney, Australia: Springer, Mar. 2017. ISBN: 978-3-319-54328-4. DOI: `10.1007/978-3-319-54328-4\_7`.

[27] William G Halfond, Jeremy Viegas, and Alessandro Orso. “A classification of SQL-injection attacks and countermeasures”. In: *Proceedings of the IEEE international symposium on secure software engineering*. Vol. 1. IEEE. 2006, pp. 13–15.

[28] Luuk Hendriks, Pieter-Tjerk de Boer, and Aiko Pras. “IPv6-specific misconfigurations in the DNS”. In: *Proceedings of the 13th International Conference on Network and Service Management (CNSM)*. Tokyo, Japan: IEEE, Nov. 2017. ISBN: 9781538621530. DOI: `10.23919/CNSM.2017.8256036`.

[29] Monika Rauch Henzinger. “Finding near-duplicate web pages: a large-scale evaluation of algorithms”. In: *Proceedings of the 29th Conference on Research and Development in Information Retrieval)*. Ed. by Efthimis Efthimiadis et al. Seattle, Washington, USA: ACM, Aug. 2006. ISBN: 978-1-595-93369-0. DOI: `10.1145/1148170.1148222`.

[30] internetarchive. *Wayback CDX Server API - BETA*. Aug. 7, 2013. URL: `https://github.com/internetarchive/wayback/tree/master/wayback-cdx-server` (visited on 04/13/2021).

[31] Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. “You shall not pass: Mitigating SQL Injection Attacks on Legacy Web Applications”. In: *Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. Ed. by Hung-Min Sun et al. Taipei, Taiwan: ACM, Oct. 2020. ISBN: 978-1-4503-6750-9. DOI: 10.1145/3320269.3384760.

[32] Andrew Kalafut et al. “An empirical study of orphan DNS servers in the internet”. In: *Proceedings of the 10th Internet Measurement Conference (IMC)*. Ed. by Mark Allman. Melbourne, Australia: ACM, Nov. 2010. ISBN: 978-1-4503-0057-5. DOI: 10.1145/1879141.1879182.

[33] Victor Le Pochat et al. “Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation”. In: *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*. Ed. by Alina Oprea and Dongyan Xu. San Diego, CA, USA: Internet Society (ISOC), Feb. 2019. ISBN: 1-891562-55-X. DOI: 10.14722/ndss.2019.23386.

[34] Inyong Lee et al. “A novel method for SQL injection attack detection based on removing SQL query attribute values”. In: *Mathematical and Computer Modelling* 55.1-2 (2012), pp. 58–68.

[35] Sebastian Lekies et al. “Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets”. In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM, Oct. 2017. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134091.

[36] Frank Li and Vern Paxson. “A Large-Scale Empirical Study of Security Patches”. In: *login Usenix Mag.* 43.1 (2018).

[37] Frank Li et al. “Keepers of the Machines: Examining How System Administrators Manage Software Updates For Multiple Machines”. In: *Proceedings of the 15th Symposium On Usable Privacy and Security (SOUPS)*. Ed. by Heather Richter Lipfor. Santa Clara, CA, USA: USENIX Association, Aug. 2019. ISBN: 978-1-939133-05-2.

[38] Daiping Liu, Shuai Hao, and Haining Wang. “All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records”. In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by Christopher Kruegel and Andrew C. Myers Shai Halevi. Vienna, Austria: ACM, Oct. 2016. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978387.

[39] Joseph Luft and H Ingham. “The Johari Window: a graphic model of awareness in interpersonal relations”. In: *Human relations training news* 5.9 (1961), pp. 6–7.

[40] Gurmeet Manku, Arvind Jain, and Anish Sarma. “Detecting near-duplicates for web crawling”. In: *Proceedings of the 16th World Wide Web Conference (WWW)*. Ed. by Carey Williamson and Mary Zurko. Banff, Alberta, Canada: ACM, June 2007. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242592.

[41] Mitre. *CVE-2014-1776*. Jan. 29, 2014. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1776 (visited on 06/07/2021).

[42] Landon Curt Noll. *FNV Hash*. Jan. 3, 2021. URL: http://www.isthe.com/chongo/tech/comp/fnv/ (visited on 06/17/2021).

[43] OverTheWire. *Natas Wargame Challenges*. URL: https://overthewire.org/wargames/natas/ (visited on 06/06/2021).

[44] Roberto Perdisci et al. “IoTFinder: Efficient Large-Scale Identification of IoT Devices via Passive DNS Traffic Analysis”. In: *Proceedings of the 5th IEEE European Symposium on Security & Privacy (EuroS&P)*. Ed. by Frank Stajano and Lujo Bauer. Genoa, Italy: IEEE, Sept. 2020. ISBN: 978-1-7281-5087-1. DOI: 10.1109/EuroSP48549.2020.00037.

[45] Walter Rweyemamu et al. “Clustering and the Weekend Effect: Recommendations for the Use of Top Domain Lists in Security Research”. In: *Proceedings of the 20th Passive and Active Measurement (PAM)*. Ed. by David Choffnes and Barcellos Barcellos. Vol. 11419. Lecture Notes in Computer Science. Puerto Varas, Chile: Springer, Mar. 2019. ISBN: 978-3-030-15985-6. DOI: 10.1007/978-3-030-15986-3_11.

[46] Quirin Scheitle et al. “A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists”. In: *Proceedings of the 18th Internet Measurement Conference (IMC)*. Ed. by Ben Y. Zhao and Ethan Katz-Bassett. Boston, MA, USA: ACM, Nov. 2018. DOI: 10.1145/3278532.3278574.

[47] Lwin Khin Shar and Hee Beng Kuan Tan. “Defeating SQL injection”. In: *Computer* 46.3 (2012), pp. 69–77. DOI: 10.1109/MC.2012.283.

[48] Raffaele Sommese et al. “The Forgotten Side of DNS: Orphan and Abandoned Records”. In: *Proceedings of the 5th IEEE European Symposium on Security & Privacy (EuroS&P)*. Ed. by Frank Stajano and Lujo Bauer. Genoa, Italy: IEEE, Sept. 2020. ISBN: 978-1-7281-5087-1. DOI: 10.1109/EuroSPW51379.2020.00079.

[49] Drew Springall, Zakir Durumeric, and J. Alex Halderman. “FTP: The Forgotten Cloud”. In: *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Ed. by Domenico Cotroneo and Cristina Nita-Rotaru. Toulouse, France: IEEE, June 2016. ISBN: 978-1-4673-8891-7. DOI: 10.1109/DSN.2016.52.

[50] Sid Stamm, Brandon Sterne, and Gervase Markham. “Reining in the Web with Content Security Policy”. In: *Proceedings of the 19th World Wide Web Conference (WWW)*. Ed. by Michael Rappa and Paul Jones. Raleigh, North Carolina, USA: ACM, Apr. 2010. ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772784.

[51] Marius Steffens et al. “Don’t Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild”. In: *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*. Ed. by Alina Oprea and Dongyan Xu. San Diego, CA, USA: Internet Society (ISOC), Feb. 2019. ISBN: 1-891562-55-X.

[52] Ben Stock et al. “From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by Ninghui Li and Christopher Kruegel. Denver, CO, USA: ACM, Oct. 2015. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813625.

[53] Ben Stock et al. “Precise Client-side Protection against DOM-based Cross-Site Scripting”. In: *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. Ed. by Kevin Fu and Jaeyeon Jung. San Diego, CA, USA: USENIX Association, Aug. 2014. ISBN: 978-1-931971-15-7.

[54] Nicolas Surribas. *Wapiti: The Web Application Vulnerability Scanner*. Feb. 20, 2021. URL: https://wapiti.sourceforge.io/ (visited on 04/30/2021).

[55] Ole Tange. “GNU Parallel: The Command-Line Power Tool”. In: *login Usenix Mag.* 36.1 (Feb. 2011), pp. 42–47.

[56] Mike Ter Louw and V. N. Venkatakrishnan. “Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers”. In: *Proceedings of the 30th IEEE Symposium on Security & Privacy (S&P)*. Ed. by David Du. Oakland, CA, USA: IEEE, May 2009. ISBN: 978-0-7695-3633-0. DOI: 10.1109/SP.2009.33.

[57] Christian Tiefenau et al. “Security, Availability, and Multiple Information Sources: Exploring Update Behavior of System Administrators”. In: *Proceedings of the 16th Symposium On Usable Privacy and Security (SOUPS)*. Ed. by Heather Richter Lipfor and Sonia Chiasson. Virtual Event: USENIX Association, Aug. 2020. ISBN: 978-1-939133-16-8.

[58] Flavio Toffalini et al. “Google Dorks: Analysis, Creation, and New Defenses”. In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez. San Sebastian, Spain: Springer, July 2016. ISBN: 978-3-319-40667-1. DOI: 10.1007/978-3-319-40667-1_13.

[59] Kali Tools. *DirBuster Package Description*. URL: https://tools.kali.org/web-applications/dirbuster (visited on 06/16/2021).

[60] Kali Tools. *Gobuster Package Description*. URL: https://tools.kali.org/web-applications/gobuster (visited on 06/16/2021).

[61] Niels L. M. Van Adrichem et al. “DNSSEC Misconfigurations: How Incorrectly Configured Security Leads to Unreachability”. In: *Proceedings of the 3rd Joint Intelligence and Security Informatics Conference*. The Hague, The Netherlands: IEEE, Sept. 2014. ISBN: 978-1-4799-6365-2. DOI: 10.1109/JISIC.2014.12.

[62] Benjamin VanderSloot et al. “Towards a Complete View of the Certificate Ecosystem”. In: *Proceedings of the 16th Internet Measurement Conference (IMC)*. Ed. by Phillipa Gill et al. Santa Monica, CA, USA: ACM, Nov. 2016. ISBN: 978-1-4503-4526-2.

[63] Philipp Vogt et al. “Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis”. In: *Proceedings of the 14th Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA: Internet Society (ISOC), Feb. 2007.

[64] Secure Code Warrior. *Secure Coding Traing*. URL: `https://www.securecodewarrior.com/products/training-ground` (visited on 06/18/2021).

[65] Ben Wolford. *Everything you need to know about the "Right to be forgotten"*. URL: `https://gdpr.eu/right-to-be-forgotten/` (visited on 06/28/2021).

[66] Jonathan Zittrain, John Bowers, and Clare Stanton. "The Paper of Record Meets an Ephemeral Web: An Examination of Linkrot and Content Drift within The New York Times". In: *Social Science Research Network (SSRN)* (2021), pp. 1–13.

# A
# List of File Extension Excluded from Probing

- .3g2
- .3gp
- .3gp2
- .3gpp
- .3gpp2
- .ai
- .avi
- .bmp
- .css
- .dib
- .eot
- .eps
- .f4a
- .f4b
- .f4v
- .flv
- .gif
- .heic
- .heif
- .ico
- .ind
- .indd
- .indt
- .j2k
- .jfi
- .jfif
- .jif
- .jp2
- .jpe
- .jpeg
- .jpf
- .jpg
- .jpm
- .jpx
- .js
- .m4a
- .m4b
- .m4p
- .m4r
- .m4v
- .mj2
- .mov
- .mp2
- .mp3
- .mp4
- .mpe
- .mpeg
- .mpg
- .mpv
- .oga
- .ogg
- .ogv
- .ogx
- .pdf
- .png
- .psd
- .qt
- .svg
- .svgz
- .swf
- .tif
- .tiff
- .ttf
- .webm
- .webp
- .wma
- .wmv
- .woff

# B

# List of Keywords for Classifying Types of Web Pages

## B.1. Redirects

- `window.location`
- `top.location`
- `http-equiv=‘‘refresh"`
- `Redirect(`
- `Object moved to`

## B.2. Page Not Found

- `404 Not Found`
- `404 Error`
- `Page Not Found`

## B.3. Frames

- `<iframe`
- `<frame`
- `<frameset`