EXTENDING HONEYTRAP WITH LUA SCRIPTING

FINAL REPORT

M.G.A. Janssen
N. van Nes
T. Oomens

DUTCH SEC

TU Delft

Preface

The Bachelor End Project, the conclusion of the bachelor's programme for Computer Science at the Technical University of Delft. Ten full-time weeks to work on a product of a real company. Exploring the field of cybersecurity seemed to be a well fitting subject with the courses that we followed for the bachelor's programme. DutchSec provided us with a concrete idea of the broad field of cybersecurity and after some discussions with our supervisors we determined that the Honeytrap project was the most interesting. What sounds better than trapping hackers on a fake server where each and every move is monitored by the software? It's like watching criminals break into your camera guarded empty house when you let the door open on purpose.

Before diving into the interesting matter of cybersecurity, we would like to thank everyone that made our project possible. These people include, but are not limited to, the people at DutchSec, especially Chief Hacking Officer R. Verhoef, our supervisor Dr. C. Doerr from Delft University of Technology and the contributors to all the open source projects that were used.

Martijn Janssen
Nordin van Nes
Thomas Oomens
Delft, July 2018

Summary

DutchSec is a security-driven company, working to enhance the security in existing systems. This is done by collecting data on machines and network activity. DutchSec takes an interest in security, big-data, red-teaming and education. Security is the main driver for the company, improving the security and making the world-wide-web a safer place.

Honeytrap is one of the main products that is aimed to fulfill this goal, providing real-time insights into network activity, potentially exploited vulnerabilities in devices exposed to the network. Honeytrap can be used in both internal and external deployments, providing threat intelligence to both the outside-world and internal infrastructure. The product provides simulated interaction with a real system or system services, avoiding detection by the attacker while still collecting accurate data.

The major flaw concerning Honeytrap is its low speed of new deployments, advancements in hacking methods used to compromise systems are made rapidly. With new vulnerabilities being found on a daily basis, a system should be able to change just as fast. This problem can be solved by using a non-compiled scripting language, removing the need to compile and deploy the entire product on every code change. With an update to the script written for a specific simulated interaction, the change can be used immediately. It also allows an easier way to add a new service or protocol by enabling the user to write complete interaction scripts in the scripting language.

This report describes the process, motivation and design choices made during the Bachelor End Project in collaboration with DutchSec. The project consists of implementing Lua-scripting into Honeytrap, which is programmed in Go. The following chapters will discuss which design choices were made, how the research was performed and how the final functionalities were implemented. A detailed system verification is done with proof of added value and besides that the system testing methods are described. Furthermore a conclusion is given that discusses what the project has achieved, what the use-cases are and whether it does what the client wants it to do.

Contents

1 Introduction

Honeytrap is a high-interaction honeypot used to trap malicious hackers and compromised systems by slowing them down and identifying them when they enter your system. For Honeytrap, a variety of services is already implemented. From telnet, SSH and FTP to more specific implementations like ethereum, mongoDB and IPP (internet printing protocol). Honeytrap is built in such a way that a new service can be added without rewriting the entire core. A new service can be written and added as separate files, after which users can choose which services to use for their Honeytrap by activating them through the configuration file.

Honeytrap is currently programmed in Go, a fast, reliable and easy to use language. It allows for easy multithreading, concurrency, deployment and communication through different channels. What Honeytrap currently lacks is support for quick (small) changes to react to new types of attacks. Every time a change is made in the source code, Honeytrap must be compiled again to apply new code. This is especially tedious when information needs to be gathered through specific responses in a narrow time window.

The world of exploits is quickly changing and evolving, with exploits being released on the regular. By implementing Lua scripting into Honeytrap, the response time for the newest exploits and vulnerabilities can be lowered. Not only does Lua increase the speed at which Honeytrap can react, it also increases the amount of ways that Honeytrap can be used to analyze incoming data or do postprocessing on outgoing data. All this in a more versatile manner.

2 Problem Definition

As described in the introduction the world of exploits, hacks and DDoS-attacks is changing rapidly, requiring a system that can change just as fast with it. While Honeytrap already gives an easier way to implement many different exploits in a convenient way, it isn't quick enough yet, since it requires rebuilding the entire project on each change/addition. Therefore quick deployment of connection and command scripts will help enormously in case a new exploit is found. These scripts will be written in Lua, which is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.

2.1. Goal of this project

The main goal of this project is to make it completely possible to alter current implementations and to implement any new protocol or exploit in less time then the current project allows. This with the goal to be able to react rapidly on zero-day exploits or implement protocols that are very specific and are only of interest for a small group of users. It should therefor be possible to alter current implementations/protocols, but also to open up ports to newly created Lua-scripts that can implement entirely new services, ie. a crypto wallet/server or a specific protocol like HTST.

2.2. The client's goal with this project

As described above the main goal for this project is to implement an easy way to add/upgrade support for different protocols/services. The Honeytrap package where this is implemented in, is however part of a bigger picture at DutchSec. They have quite some programs that help to scan for vulnerabilities and visualize obtained data, where Honeytrap's main goal is to obtain as much data as possible. At this time Honeytrap is free to use and open source. DutchSec is however working on a new project called Honeyfarm, which will be the enterprise edition of Honeytrap: A main server through which the user can easily manage/update/view all of its Honeytrap instances and agents.

While this was thought not to interfere with this project at first, it appeared after some meetings with the client that some of the proposed features could not be build into Honeytrap itself, since these would be more suitable for the Honeyfarm project, which did not fall within the scope of this Bachelor End Project. It will however give some nice options for the future of Honeytrap: Where scripts now have to be added manually or through a git submodule, these can later be managed by the Honeyfarm edition. Furthermore the AB-tests that were added and the separate statistics tool that were developed for this project can be implemented by Honeyfarm in the future, allowing for central management of all deployed instances.

2.3. Approach

There were two main points that needed researching. First, the current implementations of Lua in other Honeypots. Secondly, the different ways of implementing Lua into Honeytrap. Below different honeypot alternatives are given and following these, some different options for the implementation of Lua into Go are given as well.

2.4. Alternatives

The majority of honeypots found on the internet consist of a single purpose implementation. The honeypots listed there are only suitable for one goal, being a honeypot for a single service, protocol or product. On the paralax/awesome-honeypots repository[11], a wide variety of these honeypots is listed, with the goal of simulating services like databases, authentication, http or even an exploit. These honeypots don't give a complete solution like Honeytrap, which allows the user to have a honeypot for a great number of services in one single application.

Looking further into Lua scripting, or for completeness, any type of scripting (Lua, Javascript or something else), there isn't a single Honeytrap that implements a scripting language which is the goal for this project. Concluded was that there is no example or other off-the-shelf implementation of this feature that can be looked into or compare to. Resulting in the research question: 'What is the best way of implementing Lua scripting into Honeytrap'.

After the implementation of basic lua-scripting functionality an evaluation was conducted with Remco Verhoef (Chief Hacking Officer at DutchSec) and Christian Doerr (Assistant Professor) separately. In the first meeting with Remco, the scripting functionality was checked and the vision behind the code was shared. Remco agreed with the vision, and suggested that the core of Honeytrap could be changed to meet the vision. The change is described in this report under section 5.7.

In the meeting with Christian the progress was discussed, the vision was explained and the upcoming steps were consulted. In this meeting, the research steps within the project were discussed. Dr. Christian Doerr mentioned that there is a lack of clarity about how hackers use different services and malware to run their services. Christian showed a paper about the 'Rise of IoT Compromises', in which an IoTPot has been set-up and exploits were collected. That was the point where the paper stopped and it did not do any research about the communication and how the exploits were exploited. What could be interesting is to look at the shared code between malware: Do hackers use the same libraries, do they share code, is one hacker the same as another or are they related based on the code or libraries used? In one situation Christian encountered a DDoS attack, in which first servers were being scanned on the vulnerabilities and checked how the servers reacted, then a test attack was performed. After this test attack, a real DDoS attack was executed on these memcached servers. Following this DDoS attack some hackers wanted to create the same attach and wrote documents about the potentially used malware and converged to a single best solution and launched their own attacks following this example.

One other idea that was discussed was to collect all endpoints that were being called in the Honeytrap and predict what the result would be of these calls. This allows for better insight in the attack methods used and poll popularity for specific exploits.

Lastly the idea was mentioned to map calls that are made to the malware when a device is compromised, analyzing how the compromised devices are controlled from the command and control servers. What are the things the attacker wants to do after the system is compromised.

3 Problem Analysis

To get a clear overview of the current situation and the underlying problem, talks with Remco and Dr. Christian Doerr have been important. The first problem that was mentioned was the speed of deployment of a fix to a Honeytrap server when an exploit or a changed interaction has been found. It does not only take time to write correct responses for an exploit, but the deployment also takes time because a new build of Honeytrap needs to be created and deployed, this is not even taking into account. Besides the slow deployment process there is a second problem: versioning within different services. Intruders often use different exploits in a single attack. The exploits target different versions of the vulnerable programs as well. This means that the Honeytrap needs to be provided with a lot of scripting logic. Within Go this would remain a major hassle, because a Go program needs to be built and released when new code has been written. In the issue list of Honeytrap, scripting was mentioned as one of the probable solutions to this problem. Scripting provides a solution to the build- and deploy phase. Scripts do not have to be built and can therefore be executed directly from the original file. Secondly, the deployment and versioning of different scripts can be managed by a single central server.

Analyzing the problem went further than only talking with supervisors, working with Honeytrap itself provided the most concrete evidence for the problem. You want to be able to react instantly when new traffic is found on the Honeytrap, minimizing the amount of missed attempts from that attacker. With the full Go implementation, for every change in the method that handles the incoming connections, you have to build the entire application if any changes are made. This also requires the user to deploy and restart Honeytrap again. This is not something that you want to do every time new traffic is found in Honeytrap where no response (or an erroneous response) is given. The first week of working with Honeytrap provided worthy input for the analysis.

At the end of the first week an endpoint came forward that was suddenly requested a lot in comparison to other endpoints. A simple Google search on the endpoint gave a CVE for the endpoint. A CVE is described as Common Vulnerabilities and Exposures. This is a list of vulnerabilities and exposures that are commonly known and researched by different organizations. Often there is a detailed description about the CVE and how hackers might exploit the flaw in the system. The CVE that was found in combination with the endpoint showed a vulnerability in GPON routers. Using two HTTP requests hackers could get access to the admin environment without having to log on. The first endpoint was used to make unauthorized calls in the system, the second was used to display the results of the calls made in the first endpoint. When hackers can get admin access without logging in they can plant their own software on the routers without the knowledge of the system owners.

The publication date of the CVE and the observed growth in calls on the endpoint were not more than 2 days apart. This shows that hackers might follow the CVE list and if a new CVE is published, use this hack to test known IP addresses whether they contain the vulnerability. A quick response on these vulnerabilities is therefore necessary to catch the hacker in the middle of the act and retrieve valuable information. The retrieval of these valuable resources sometimes have a limited time frame. The URLs that were used to download binaries to the target were most of the time already offline when visiting these the morning after an attack. The hackers that were calling the GPON router endpoints had a URL within their payload to exploit the GPON vulnerability. This URL was offline for 36 hours. Suddenly the URL was online for about 15 minutes, our Honeytrap was being exploited. So within 4 days of the publication of the CVE, hackers already had an exploit ready to be deployed to all GPON routers.

This means that speed is key when responding to new vulnerabilities and exploits, especially in cases where hackers use more sophisticated exploits with a combination of steps to complete the end goal. This is were the current problem of Honeytrap arises. Maintaining, building and deployment of the Honeytrap is too big of a hassle currently. Scripting provides a way in which speed is maintained when responding to new exploits. Hackers are more likely to go further with their exploits when they get the correct responses on their requests. When the Honeytrap can, for example, imitate a slave of a command and control server, the most valuable information about the hacker can be retrieved: the communication within the network of a botnet.

3.1. Working methods at DutchSec

DutchSec maintains a different mentality than most companies. They want to employ hackers, not software developers. This is the way they want to stimulate creativity inside DutchSec as a company, providing everyone a free and open environment to work in. If a problem needs solving, the first step is to find the most simple solution and work from there. You can't come up with the final solution in one go, start with a prototype and improve upon this in later stages. This is a more free approach to scrum and agile development, which fits the company at this age (1 year) quite well. DutchSec can be seen as a very nimble company because of this mentality; building a variety of products and still being able to swiftly adapt to the customers' or markets' needs and requirements.

3.2. Impact of Honeytrap on clients

A hack can be started in different ways, in essence there are two different approaches. An external hack and an internal hack. A hack is defined as gaining unauthorized access to data in a system or computer with the intention to read, retrieve or modify the data. An external hack comes from outside the company, and can be monitored by Honeytrap on open external services like websites, apps and FTP services. Potential hackers will be caught within the web of the Honeytrap, their activity will be monitored and appropriate action can be taken on the external services. Internal hacks are a bit different than external hacks. Internal hacks are executed by employees or externals that require access to internal systems to do their job. More and more stories are coming up about hacks that combine social engineering, real job applications and externals that gain unauthorized access to data from within the company. These types of hacks are currently one of the most invisible types of hacks that exist in the world of information systems. Honeytrap can be the saviour in these types of situations. For example, running an internal Honeytrap can trap internal hacks by catching false log-ins, or unwilling data calls to internal web-servers or file-servers. When the unauthorized access has been logged appropriate action can be taken against these actions from internal hackers. Therefore there can be a lot of significant impact on external and internal security when clients will implement the Honeytrap product into their services.

Not only customers of DutchSec can benefit from the impact of Honeytrap, also the whole computer science community will benefit from the software. How will the community benefit from the Honeytrap product? As said in the introduction exploits are quick, sometimes very quick. The sooner a new exploit is spotted or the sooner a patch is written to close a potential exposure or vulnerability, the better the outcome will be for a lot of people, and the harder it gets for potential hackers to start their operation. A lot of websites and applications are dedicated for sharing information and data on potential exploits (VirusTotal,), unsafe URLs (VirusTotal) and (un)known exposures and vulnerabilities (CVE lists). Another valuable source is MetaSploit, which implements a vast collection of vulnerabilities. These implementations can be used to find the ways that vulnerabilities can be exploited. Most of the information on these websites is coming from already running Honeytraps or logging systems from around the world owned by different companies and hobbyists. With the potential growth of the product, the community will be provided with a boost of information.

3.3. Development procedure

As described in the previous section, the working method in DutchSec closely resembles agile. This is why the chosen development methodology is agile. After every meeting held with Remco Verhoef or Dr. Christian Doerr, the direction at that point was re-evaluated and tweaked.

3.4. Honeytrap implementation

Honeytrap is a system that is unlike other Honeypots, instead of focusing on a single solution, multiple solutions are wanted in a single product. The system is designed with configurability in mind, there are no single ways to do something. This makes Honeytrap very usable in different applications and for different purposes. Recently, plugin support was added to Honeytrap, allowing users to write plugins for Honeytrap to add specific actions or transformations to the data at specific points in the code.

Honeytrap consists of multiple major components. In the following sections, the components will each get a brief description on how they work and what functionality they provide.

3.4.1. Listeners

An attacker has various ways to infiltrate a system. This is why Honeytrap allows for different listening modes. There are several modes:

- **Socket**: The socket listener allows binding to specific ports on the device, not interrupting with other services on the device
- **Canary**: Sends a SYN-ACK to every SYN it receives and logs every incoming payload
- **Raw**: Allows implementation of a custom network stack
- **Netstack**: Custom network stack implementation in Go

The scripter-solution is built around the socket listener, which will eventually bind the Lua script to a port, or for the generic Lua scripter, listen to all (or a range) of the ports and give responses when it is able to.

3.4.2. Services

A wide variety of services is implemented in Honeytrap, all of which can be enabled using the `config.toml`. Each service is registered at startup, after which the services get their dependencies attached. The service handles interaction from the moment the connection arrives at Honeytrap. When the connection arrives, each service is checked by calling the `canHandle` method, if the service is capable of handling the method, a new goroutine is created. A goroutine is a way of creating a thread for a function in Go, the called function is added to the internal Go scheduler. This is one of the strengths of Go, which has a scheduler that contains the cpu-cores from the host machine, and will schedule all goroutines on these cores, allowing for easy multithreading by just adding the `go` statement before a function. This allows Honeytrap to have a different thread for each connection, resulting in low-latency responses and fast operation.

The `go` call to the `handle` method of the service manages the entire connection. The function will continue until either an error occurs, the connected client closes the connection or when Honeytrap has successfully responded to the attacker. In the `handle` method, the full connection is handled, from handshake to interaction and closing the connection again. Because the `canHandle` method only passes connections that can be handled by the service on to the handle method, this method only contains the code for handling the traffic it will eventually receive. Because of this, the specific Go implementation for the protocols can be used to handle the connection, and create a clear, single thread that runs for every connection.

3.4.3. Directors

An attacker reacts on a specific response to his submitted commands. A way to give the correct response as if the attacker is connected to a real system can be done with an LXC container. This is a Linux container based on a Linux image, like Ubuntu. If an attacker connects to Honeytrap, the connection can be forwarded to the container, which functions like a real system. Honeytrap is built in such a way that makes it easy to add different types of directors. LXC, QEMU, gVisor and forwarding are currently implemented.

3.4.4. Channels

A channel is a queue of messages that contains all the data from the different services. All Honeytrap components that have a channel can push events to the channel. These events mostly contain information from incoming connections. Honeytrap now contains one channel for logging, but this could be expanded at a later point for more configurability. In the `config.toml`, a filter can be specified. This filter contains several pushers, which will listen to the channel.

3.4.5. Pushers

A pusher is the location where data is finally written to. All data in the channel is fetched by each of the pushers and stored in their corresponding location. A pusher can be an Elasticsearch cluster, Apache Kafka, a file on disk or console output. Again, just like the rest of Honeytrap, these pushers are expandable and new ones can be easily created.

3.5. Tools

Multiple tools, programs and protocols are used to enhance Honeytrap and the way of adding functionality to the current implementation. The ones that are most relevant are given below:

3.5.1. Hosting

To test the Lua scripting implementation on a live Honeytrap, the Honeytrap server was deployed to a DigitalOcean droplet (2GB RAM, 1vCPU, 50GB storage, 2TB transfer) accompanied by Elasticsearch and Kibana to allow for storage and easy viewing of the data. To this Honeytrap server, several honeytrap-agents were set up to allow for more incoming connections on a wider range of IPs instead of just one. This is done by configuring the Honeytrap server as an agent-listener and setting the port that is used for forwarding the agent connections. Additionally, two domains (.com and .nl) were forwarded to one of these agents.

3.5.2. Git/Github

Honeytrap is open-sourced through Github. Any addition that is to be made to Honeytrap itself is done through pull requests in a fork on Github. One should make a branch in which the changes are placed and after a successful peer review, a working Travis build and good enough code coverage, the additions can be merged into the main branch.

3.5.3. TOML

Honeytrap uses TOML to store config data, this allows each user to create a configuration that suits their needs: Users can add the services they would like and add the channels to which gathered data is sent. Lua scripting adds the following to the config: a scripter for a service and an abtester to use in the entire Honeytrap. There are a few different types that can be added through the config of which the most convenient are listed below:

- **port:** Adds a listener to a port and connect a service to it.

  ```
  [[port]]
  port="tcp/80"
  services=["http"]
  ```

  The port-field specifies the port that is bound with which protocol. The services-field gives a list of services that are enabled on this specific port, allowing the user to bind one or more services. This allows Honeytrap to find the specific service that is able to handle the specific requests coming in on that port.

- **service:**

  ```
  [service.ssh-simulator]
  type="ssh-simulator"
  scripter="lua"
  ```

  The part after the `service` statement gives the name of the service, also to be used in the port config. The type indicates the service declaration in Go. The scripter attaches the declared scripter to the service, allowing scripter usage.

- **abtester:**

```
[abtester]
file="abtests.json"
```

  Add an abtester and specify the location where the abtests reside in.

- **scripter:**

```
[scripter.lua]
type="lua"
folder="scripts"
cleanupTimer="1"
```

  The type-field matches the scripter declared in the project to the name specified in the `scripter.lua` name above, this name is used to link the scripter to the services. The `folder` locates the lua scripts on the filesystem. The `cleanupTimer` is the interval at which a cleanup task is run to avoid using too much RAM for all the Lua states on the system.

- **channel:**

```
[channel.console]
type="console"

[channel.file]
type="file"
filename="honeytrap.log"

[channel.elasticsearch]
type="elasticsearch"
url="http://127.0.0.1:9200/honeytrap"
```

  The first line again indicates the name of the component, the `type` is selecting the channel for logs to be used. The other options can differ based on the type of channel, like a filename or a url.

- **filter:**

```
[[filter]]
channel=["console", "file"]
```

  The filter line allows the user to use only specific channels to output to, allowing more fine-grained control for the output. This could later be expanded to allow specific services to only log to a specific location.

3.5.4. Elasticsearch

Honeytrap is a tool that tries to collect as much data from incoming attacks as possible. While this data can be stored in multiple ways, the most convenient one is Elasticsearch. A database like system that can store a vast amount of data and create easy ways to access them later in a fast way. For this project the API of Elasticsearch is used for the custom stats tool which displays the different incoming IPs, connections, sessions and messages. More about this custom stats tool, which can be found in the findings chapter.

3.5.5. Kibana

While Elasticsearch can store a lot of data, it has no good way of visualizing it, it's only used for storage. This is where Kibana comes in, a web interface that has multiple options of displaying data stored in Elasticsearch, developed by the same company. The most used feature for this project was the 'discover' feature of Kibana which gives a stacked overview of the messages in a specific time interval. These messages are expandable, showing all content of these messages. This made it quite easy to discover new types of attacks and where to focus on. Besides that it gave a good view on the results of newly added implementations. Kibana has many other features to visualize data, but most of these required some sort of payment/subscription that was not available for this project.

4 Design

4.1. MoSCoW

What can be done with the scripting implementation of Honeytrap? Attackers want a successful response on every request during their exploit. This means that they perform multiple calls to a certain endpoint or within a shell, as a sort of test. In the paper on IoT attacks[13] are described with several tests that the attacker does before running its exploit. It starts with a simple print statement within telnet and checks whether the exact string is printed back to the console. After the initial test it performs a random non-existing command, which is checked on the right error that needs to be returned. After these tests, the exploit starts. With each task the exploit tests whether its action was successful, otherwise it aborts the attack. The last part is exactly that what must not happen. The most interesting part of the attack is the lead towards the attack, how the attacker finds its targets and what the exploit does. Every time an attacker quits its attack it is less likely that he will return to perform its attack again and that the Honeytrap can collect the important data.

What can be done with the collected data of Honeytrap? If the attack is successful and the data has been collected on the lead and the execution of the attack, important information can be extracted. Information like the vulnerabilities that the exploits used, or even the origin server of the attack or the writers of the software. Or collect information on matching exploits, that use the same code or library to perform their attacks. Via this way a map could be made of possible vulnerable systems, services and devices, and thereby a map of attacks, attackers and link between different exploits. Not only attackers can be caught in the act, but also whole organizations that perform the attacks. Linking to even countries, governments or organizations. Counterinitiatives can be built on the collected information or organizations can be assisted to prevent these intrusions to happen on their systems.

Based on the previously mentioned criteria a MoSCoW analysis can be made. The analysis determines which features must be implemented into the system, which features should be implemented into the system, which features could be implemented into the system and which features won't be implemented into the system within the timeframe of our project. Below a list can be found of all features that can be implemented, ordered with MoSCoW:

MoSCoW	Feature	Description
Must have	Lua-Go implementation	Lua is not standard supported by Go, a library must therefor be integrated that adds this support
	Scripter object	An objectlike interface/structure must be created that can be loaded by honeytrap and added to the differnet services/generic scripter
	Add scripter to existing services	It must be possible to add a scripter to an existing service, so that it can be extended through Lua scripting
	Add a generic scripter	There must be a new kind of service that can implement a protocol/exploit by only making use of Lua scripts
	Configuration	The scripter must be configurable through the config.toml
	Handle method	A Handle method must be available that will handle the incoming connections just like in Honeytrap itself
Should have	Web Documentation	Documentation on how to add a scripter, configure the scripter or write Lua scripts, should be added to the web documentation of Honeytrap
	Scripter between different services	It should be possible to have the same scripter connection that can be used for multiple services accessed by the same person
	Clean old scripter connections	There should be an automized tasks that cleans old connections that haven't been used for a long time
	Expose Go function to Lua	There should be some methods written in Go that are exposed to Lua: Getting the ip, logging a message, etc.
	CanHandle method	A CanHandle should be available that will return whether the script can do anything with the incoming connection
	Writing to connection from lua	It should be possible to write over the connection using Lua only
Could have	AB-Tester	An AB-Tester could be implemented that returns different values for certain keys.
	Support other scripters, ie js.	It could be usefull to build the scripter in such a way that not only Lua, but also other languages can be implemented easily in the future.
	Download found files	A method that can download files from found urls in Lua could be added
	Reload scripts button	A button that can reload all scripts without restarting Honeytrap could be added
	REST-support	Support could be added for REST-type calls, so that the user can fetch and send HTTP/REST calls from within Lua scripts.
	Web implementation	A web implementation for the scripter in the website of Honeytrap could be implemented
Won't have	Separate website	A separate website would have been nice, in which one could manage all of its scripts

4.2. Visual

The current implementation of Honeytrap does not provide any kind of design except for a minor website, implemented in React, that shows some information on currently gathered data like sessions and connections. This is because almost all visualization is handled by Marija [3] and Kibana, the first being a custom designed tool by DutchSec to display data in a graph-like manner, and the latter a third-party software tool. For this project there are however some visualizations, specific for the Honeytrap, required. The current idea is to implement the different scripts in a plugin-based manner, allowing the user to add, activate or delete scripts. There should be an overview on the generated website where all the current (active) scripts can be viewed and where they can be managed. This page (or pages) will be placed on the Honeytrap dashboard to further integrate the Lua scripting into the product.

4.3. Design choices throughout the project

The first two weeks consisted of doing research. The project description stated that Lua should be implemented into Honeytrap so that it would be possible to implement new services and exploits in an easy manner. With this vision it was decided to use Gopher-Lua, a Go-Lua implementation, and build a scripter object that could be added to services of Honeytrap. The decision was made to first start with implementing Lua into existing services and when that would be working, implement a service that could process an entire message through a scripter.

This seemed to be quite easy as it took only about two weeks to implement the service extended version and even catch the first virus-binaries that were downloaded via a GPON-exploit that was discovered during the third week of the project. This was however a very basic implementation, but good enough at that time. After a meeting with the supervisor the conclusion was made that the implementation would most probably be done faster than the given 8 weeks. The decision was therefor made to choose a common type of attack after the implementation was done to measure the impact and the use of the newly implemented features.

First however a generic scripter was implemented that allowed the implementations of complete new protocols/services, the user could create a new Lua script that is able to receive an incoming connection and reply on it, through Lua-scripting alone.

Next to these core methods some extra functionalities were built in: An AB-tester, a web-extension for a script reload function and a loop that cleans unused connections. Also a separate statistics web-interface was written to parse the data in elasticsearch in a way that was more useful for the research of this project than the visualizations of Kibana. These implementations were more difficult than expected and the use of Travis and Codecov gave quite some problems towards a correct git-use. The possibility to do an indept research on the usefulness of the additions to Honeytrap of this project became therefor less interesting. A meeting in week 7 also gave us the knowledge about an enterprise edition of the software that was being developed, which meant that many of the design-plans that were created at the beginning became redundant, since the client didn't want them in Honeytrap anymore, but in this new Enterprise Edition: HoneyFarm. The choice was then made to only create a simple web interface that showed some more summarized information about incoming connections to see if a new type of attack could be easily recognized.

Besides this, extensive testing was added and the web documentation of Honeytrap was extended with explanation on how to implement the scripter and how to write your own scripts. A meeting was then held with the CHO of DutchSec in which he gave some insight on what he wanted with Honeytrap, which resulted in the start of a rewrite on the core of Honeytrap which would allow for better switching between services, scripters and directors. While this rewrite was started it has not been finished completely. The choice was therefor made to leave it outside of the pull-request of this project and create a separate Work-In-Progress pull request and a recommendation within this report.

4.4. Quality control and testing

The project required a development plan in which the quality control and testing would be guaranteed. Via this plan most of the flaws in the written software would be tackled in the development phase. Fixing bugs is less costly within the development phase and as a direct follow-up of writing a part of the software when the idea behind the piece of software is still fresh in mind of the developer that releases the source code. To maintain the quality control and testing, a combination of different tools would be used. These tools include Git, Github, Travis CI, Go testing and Codecov. Git and Github are the tools that are being used for version control, issue management and supervision purposes. Travis CI is used for the setup of an automated build process which checks whether the build is correct for different software versions and platforms. Go testing and Codecov are tools that keep the testing of the software on a sufficient level throughout the project.

First a fork of the main Honeytrap project on Github was made and placed in a separate HoneyBep repository. From this fork all issues, branches and pull requests were to be made. This has been done to minimize the load on the main Honeytrap project on Github and keep the project separate. If bugs need to be fixed or features need to be made on the main Honeytrap project, an issue would be made on Github and based on that issue a pull request was uploaded to help the overall project. All issues and pull requests considering the scripting module would be inside the fork of the project. To keep an initial control on the quality each feature or fix was provided into a new branch. From this branch a pull request could be made, and a few steps would happen before the piece of software is accepted.

The steps of quality control and testing are the following: Travis CI picks up the pull requests and tests the automated build on 4 different Go versions, namely the master, tip, v1.10 and v1.9. After Travis CI would successfully build the project, Codecov would run all tests within the project and check if there are any differences in code coverage of the tests. In principle the coverage can not decrease with a pull request, all newly written software had to be tested. After a successful test phase, the last human test would be performed. This human control and test is obligated via Github, a pull request can not be accepted without at least one approving review. The review is done by pulling the branch from the online environment and check the code on the purposed goal of the branch. Often the code, comments and tests will be discussed orally before the pull request would be merged into the scripting branch. The scripting branch is the main branch of the project of Lua scripting within Honeytrap. The discussed steps will guarantee the quality control and testing within the project.

4.5. Implementation

The implementation of the required features can be performed by activating scripts via an additional command, as a service or as separate element (scripter). After looking through the code, a few possible ways of implementing Lua scripting into Honeytrap remained:

- Implement Lua directly into the core of Honeytrap making it a part of the main product.
- Create a plugin for the recently built plugin feature for Honeytrap, allowing the user to decide whether to use Lua scripting.
- Create a separate service that will check if it can handle an incoming message and if so, process it and give a return, optionally being placed before every service in Honeytrap
- Create a separate element in the Honeytrap core that can be included and connected to existing services, allowing the user to choose which services and systems can use extra Lua scripts.

After looking into these options the first option was discarded since it should not always be required to have Lua scripting available. The plugin option was also discarded because services would need to be changed to implement scripting; not installing the plugin could create problems when a service implements scripting anyway. The discussion then came to creating a service or a separate element. While a service seemed the better option at first, it was fast discovered that this would create quite some dilemmas in implementation. Honeytrap goes through all of its services and picks one to handle the incoming message. The idea of the Lua scripts however is to react only on specific calls and this way of working would force a Lua script that implements an SSH connection to implement the SSH-handshake as well, because the SSH service would never be called. This would be the case for most of the current services, requiring Lua to re-implement all the logic that is already present in current services. Therefore the choice was made to create both a separate element which will be called ‘scripter’ that can implement external scripting languages into current services, but also create a special service that would implement new services purely by using the scripter. This way, option three and four were combined. Initially only the Lua language will be supported, but keeping in mind the option to implement others as well, for instance JavaScript, the scripter will be built in such a way that it can support other languages in the future as well.

4.6. Lua implementation

Gopher-lua implements Lua scripting in Go, providing a user friendly API in Go to integrate Lua scripting into Honeytrap. This Lua implementation makes use of a stack for reading returned variables and writing function parameters. In comparison to other Go-Lua implementations, Gopher-lua is the more user-friendly solution by creating methods that allow the user to omit having to use the stack.

The Lua scripts have to hook to specific places in Honeytrap itself, exposing an API for public functions to integrate into each other. These functions should also be easily expandable, so some methods are needed that can set these functions. These functions should contain the functionality to add additional logs, set variables in the Lua script to be used for processing the requests and so on. With all these functions, there should be no difference whether the service is implemented in Lua or Go, since everything is or can be exposed to both. If some function is missing, it is preferred to only write a specific function in Lua, or implement a function in Go using a library, it should be possible to do so in a logical and easy manner. When implementing these functions, scoping is something of importance, since some functions have properties only relevant to the connection, while others are more generic.

It would be beneficial and best to have a single state in Lua for storing these methods, attaching them to each newly formed Lua state and making them available for use on the connection, unfortunately, none of the Gopher-lua methods seems to support this, so it is needed to bind every function again from Go to the Lua state for that connection.

4.7. How to implement a Lua script

The newest version of Honeytrap already has something of the form of plugins, yet Lua also has this plugin look and feel. There is a folder that contains all Lua scripts. This folder contains a folder for each type of service and for the lua-generic service. In each of these directories Lua scripts can be placed. An extra directory can be placed in each service directory in which scripts can be placed that can be used by the other scripts, this directory should be called 'utils'.

When one wishes to add a new Lua script, a few steps need to be taken:

1. If no folder for the service that it extends exists yet, create the directory.
2. If the script requires other scripts, add these to the 'utils' directory.
3. Create a Lua script ending on '.lua' and add it to the directory as well.

After a file for the script is created, the file needs to be edited to contain a handle and canHandle method, the canHandle being a method that receives the peeked connection and returns true or false and the handle being the method that handles the complete incoming connection. Both methods can make use of the methods that have been made available from Go as specified earlier. It should then return either a value that can be used by the extended service, eg. the body of an http response, or `_return` when it has handled the message itself and doesn't want the Go-code to do so as well.

When the scripts and its required scripts are created and placed in the correct locations, the service that needs extending needs to be changed in the `config.toml`, as described in the implementation chapter in detail, an extra line beneath the service needs to be added that tells Honeytrap to connect a scripter to the service. After this a restart of the server is required to make the server load the scripts. If the service already had a scripter activated, changing the config and reloading the server is unnecessary, one can simply go to the webview of Honeytrap and move to the scripter tag, where a 'reload scripts' button is located.

5 Implementation

After the research phase the implementation of Gopher-Lua into the application and the implementation of endpoints throughout the Honeytrap application to call Lua scripts started. This was implemented as a separate scripter element and a generic service which will be explained below.

5.1. Separate scripter element

A directory was added called 'scripter' containing:

```
- lua
| - connection.go
| - lua.go
- connection.go
- methods.go
- scripter.go
- web.go
```

- **scripter.go**
  This is the main file that adds a register method to initialize itself, register scripters to other services, add some global methods and to define the different interfaces used by the scripter.
  Three different interfaces are introduced:
  - **Scripter**: While Lua is the goal for this project, the setup is done in such a way that other types of scripts are easily implemented by implementing a structure with this interface. This interface adds methods for getting a scripter-connection, getting the scripts and checking whether one or more of the scripts can handle an incoming call.
  - **ScrConn**: An interface that implements methods to work around an incoming connection, adding scripts that can be used for the type of incoming connection or service type and run these scripts to interact with data or the connecting instance.
  - **ConnectionWrapper**: A single ScrConn can have scripts for multiple services. Since we don't want to give too much logic back to the service, each service receives a ConnectionWrapper that contains the ScrConn for the incoming IP and methods that have impact on this connection wrapper. This way there is one wrapper for each service, and only one ScrConn that can work with all these different connections. This is useful when for instance an SSH-connection fetches a fake http-passwords file from the server, after which one might want to check if that same user connects over http with the given passwords.
- **connection.go**
  Implements the ConnectionWrapper interface and sets the variable for the connection and ScrConn as well as the methods to handle messages or set functions in the scripts.
- **methods.go**
  Sets functions that can be used in every script, the following methods are implemented:

  - **getRemoteAddr**: Returns the IP address of the connecting client
  - **getLocalAddr**: Returns the IP address of the Agent
  - **getDatetime**: Returns current datetime
  - **getFileDownload**: Downloads a file from a given URL
  - **getAbTest**: Returns a random value for a given key, fetched from the Abtest interface
  - **doLog**: Logs an incoming command by type and message
  - **channelSend**: Sends a message to all connected channels, ie. elasticsearch
  - **getFolder**: Returns the script folder in which the script is currently working

- **web.go**
  Honeytrap has a web interface as well that shows the different incoming connections. To implement features that influence the scripts, some extra methods are introduced in this file. The react frontend can connect with these methods and reload, add, update, delete or list running scripts.

- **lua (dir)**
  While the root dir of scripter implements the basic methods for each scripter and all interfaces for the different scripter types, the lua scripter is implemented in two files in the lua-subfolder:

  - **connection.go**
    Each computer that connects to the honeytrap is sent to one of the connected services of Honeytrap, these services can get a scripter-connection that implements scripter functionality. One connection is created for each IP-address, upon reconnecting of the same IP-address, the same connection object is returned. This file implements all methods from the ScrConn interface that are specified for the connection with Gopher-Lua. For each used script in the scripts directory a LuaState is created and stored in a 'luaConn' struct, that will be used when Handle or CanHandle is called.
  - **lua.go**
    This file contains the 'LuaScripter' struct that implements the 'Scripter'-interface mentioned above. It implements a 'New' method that lets the user create a new LuaScripter instance and implements some basic methods to initialize itself, and methods to create lua-connections and a CanHandle that checks whether lua can do something with an incoming connection. When initializing, all variables are set and each lua script available is made into a lua-state and stored. When the CanHandle method is called this struct will go through each stored lua-state and call the canHandle on it to check if at least one lua-state can handle the type of incoming connection.

5.2. Other additions to Honeytrap

While the implementation of Lua was the main goal of this project, some additions have been made to Honeytrap itself that are/can used by the scripts that can be added by the scripter. These additions are mentioned below:

- **AB-Tester**
  In the research phase it was seen that attackers often used commands like 'uname -a' or 'ls'. While it was easy to implement a script with a switch that would return specific implemented results, a feature to return a random value based upon a certain key was missing in Honeytrap. The decision was therefor made to implement this as well, resulting in the AB-Tester. The AB-Tester is a single file in abtester/abtester.go that implements an interface and struct in the same way as the storage of Honeytrap. To use an abtester a call upon the 'Namespace' method is called where a name can be passed along to create a unique abtester. The returned object uses a storage to store the different abtests available divided over groups. A namespace consists of a list of groups, with in each group a list of keys and for each key a list of values. ie. The group 'ssh' can store the keys 'uname => linux, windows, mac' and 'ls => passwords.txt, document.word, important.pdf'. Besides the data each object has a few methods that grant the ability to:

  - Set a whole set of ab-tests from a json-file.
  - Return a random value for a specific group and key

- Return a random value for a specific key
- Set a value for a specific group and key
- Set a value for a specific key

5.3. Adding a scripter to a service

To use the scripter inside an existing service a few steps need to be taken. First the configuration, if this is the first time implementing scripters a config rule needs to be made to enable scripting, set the type of scripter and set the folder that the scripts are in. An example for lua scripting:

```
[scripter.lua]
type="lua"
folder="lua-scripts"
```

The first line sets the name of the scripter service, which can be used in the services (explained later). The type is nothing else than the language the scripts are written in. At this point only the Lua-language is supported, but when more languages are implemented in the future, they can be defined here. The folder tells the program where the scripts are located on the local system.

To enable the Go-Lua implementation for a specific service, an extra rule needs to be added to that of a service:

```
[service.http]
type="http"
scripter="lua"
```

In the example the HTTP service is defined with an extra element for the scripter. This line tells the HoneyTrap that the Lua scripter is required for the HTTP service. After this step the configuration is done.

While the configuration is setup correctly there still needs to be some support in the service file itself. This should only be done once and once in the Honeytrap repository will not be required anymore.

```go
type httpService struct {
    httpServiceConfig

    scr scripter.Scripter
    c pushers.Channel
}
```

Figure 5.1: The struct for the service is extended with a scripter

```go
func (s *httpService) SetScripter(scr scripter.Scripter) {
    s.scr = scr
}
```

Figure 5.2: The SetScripter function is added to perform the addition of the scripter

```go
// Http is a placeholder
func HTTP(options ...ServicerFunc) Servicer {
    // other initialization information

    if err := s.scr.Init("http"); err != nil {
        log.Errorf("error initializing http scripts: %s", err)
    }

    return s
}
```

Figure 5.3: The Service constructor is extended with the init function of the scripter. The init function loads all scripts in the 'http' folder.

```go
func (s *httpService) Handle(ctx context.Context,
                             conn net.Conn) error {
    sConn := s.scr.GetConnection("http", conn)
    // connection handle function
}
```

Figure 5.4: The Handle method is extended with a method call that returns a ScrConn object for the specific incoming connection, with the correct 'http'-scripts loaded. This connection-object can be used to call functions, register functions or get parameters from Lua.

```go
sConn.SetStringFunction("getRequestURL", func() string {
    return req.URL.String()
})
sConn.SetStringFunction("getRequestMethod", func() string {
    return req.Method
})
```

Figure 5.5: Each service can set specific functions that will be available in the scripts, http for instance can set methods that return the requestUrl or requestMethod.

```go
responseString, err := sConn.Handle(string(body))
```

Figure 5.6: A service can call the Handle method on a ScrConn to parse an incoming message, a string is sent towards the handle method and a string is returned as well which in turn can be returned to the connecting client.

The logic for the Handle part can be different depending on the service. The service is not only bound to the script for handling the connection, but can also be used as a logger.

5.4. Adding a generic scripter as service

When there is no Honeytrap support for a specific service or protocol yet, the user can now implement a generic scripter service. The Go code for this service is already implemented and the user will only have to write a Lua script and edit the configuration. The configuration should be edited in the following way:

```
[[port]]
port="tcp/51235"
services=["generic"]
```

Where tcp/51235 is the port to which the newly added service should listen. There are two types of protocols available: tcp and udp, which should be followed by a / and the port. Next, the user should add a script in the 'scripts/generic' folder, containing a canHandle and handle method. The canHandle method receives the peeked connection and should return true if it can handle the connection:

```lua
function canHandle(request)
    return request.port == 51235;
end
```

The handle method should contain the functionality to parse the incoming message and either write a message back over the connection or return a message that Honeytrap itself will write back. A generic script has a two extra exposed methods that can be used:

`getRequest()` : When the implemented service is a HTTP like service, the incoming message can be fetched as a http-request, containing the method, header, host, form and body. When calling getRequest, the method reads the connection and parses that data to the given fields and passes these back as a JSON object.

`restWrite()` : When you need to write a HTTP-like message back to the connecting client, this method can be used. Three parameters should be sent along: The status, the response data in JSON form and a list of headers. The method then creates a HTTP request containing the correct body and headers and sends it to the client.

Furthermore there are two ways to sent back a message to the connecting client:

Let the script handle it. When the user wishes to send a message to the client with the script itself he/she can do so, either by implementing the connection in the script, or by using an exposed method. To prevent Honeytrap from trying to return a message after the handle method of the script, the script should then return "_return", telling Honeytrap that the script has managed the reply.

Let Honeytrap handle it. If the user only wishes to create a message, but not the handling of the connection itself, the script can simply return the message to be sent back to the client through the return of the handle method.

5.5. Testing

While Honeytrap itself is written in Go and the implemented scripter object as well, many of the added scripts are written in Lua and add Lua-specific implementations. The tests added needed to not only test the Go implementations, but also the lua scripts.

5.5.1. Scripter

The scripter is a complex structure which makes the testing inevitably difficult. The scripter holds a connection wrapper, a scripter connection and the scripter itself. Separate methods within these elements can be tested with certain accuracy, because these methods have a single purpose. The goal has been to separate methods based on their goal and keep the goals as separate possible to keep testing accurate. The real difficulty lies within the umbrella methods and edge cases like errors caused by vendor packages. In the end around 78% of the scripting module has been tested within Go.

How to test the different components and inner workings is the real question. Normally unit testing is supported by mocking, which means that an object to be tested depends on other complex objects. These other complex objects are separately tested and when testing the main object, the dependencies do not to have be tested again. This is why these depended objects can be mocked into the main object for testing purposes. The behaviour of the main object will be isolated and can be tested accordingly. This is useful if the real objects are impractical to include into the unit test. Go does support mocking via a separate library, but the inner workings of Honeytrap provides dummy objects. Where each method does nothing, and returns the expected outcome. The use of dummies replaces the function of mocks, the dummies are what has been used to test the inner workings of the software. An example of a dummy is the following of a connection structure:

```go
type dummyConn struct {
        conn net.Conn

        //List of lua scripts running for this
        //connection: directory/scriptname
        scripts map[string]map[string]*lua.LState

        abTester abtester.AbTester

        connectionBuffer bytes.Buffer
}

//GetConn returns the connection for the SrcConn
func (c *dummyConn) GetConn() net.Conn {
        return c.conn
}

//GetAbTester returns the ab tester for the SrcConn
func (c *dummyConn) GetAbTester() abtester.AbTester {
        return c.abTester
}

//SetStringFunction sets a function that is available
//in all scripts for a service
func (c *dummyConn) SetStringFunction(name string,
getString func() string, service string) error {
        return nil
}
```

This dummy connection can be used to test the methods in the scripter interface which make use of a connection normally. But the connection is a different type, like a Lua connection. The scripter can not be tested with a real Lua connection structure because an error would be thrown on a cyclic dependency within Go. The dummy connection made the testing of the methods possible without the care of the inner workings of a normal connection structure. A dummy scripter helped to test the registration method of a type of scripter, because the register method is independent of the type of scripter. The following test made use of the dummy within the scripter tests:

```go
// TestRegister tests the register of a scripter
func TestRegister(t *testing.T) {
        Register("dummy", Dummy)

        scripterFunc, ok := Get("dummy")
        if !ok {
                t.Fatal(fmt
                        .Errorf("unable to retrieve scripter function"))
        }

        if _, err := scripterFunc("dummy"); err != nil {
                t.Fatal(err)
        }
}
```

The tests of the scripter included the initial setup phase, the separate tests of the connection wrapper and connection methods and the inner working of the different elements. After the scripter has been tested, the tests go further into the specific Lua scripter implementation. These tests are the most valuable for the project, because these tests include the main functionality of our main goal. To implement Lua scripting into the Honeytrap framework. The following test includes multiple functionalities of the internal flow within the scripter:

```go
// TestLuaConn_SetStringFunction test the string
// functions from Go into Lua
func TestLuaConn_SetStringFunction(t *testing.T) {
        conn := ls.GetConnection("test", client)

        err := conn.GetScrConn().SetStringFunction("parameterTest",
        func() string {
                return "test"
        }, "test")

        if err != nil {
                t.Fatal(err)
        }

        result, err := conn.GetScrConn().Handle("test", "test")
        if err != nil {
                t.Fatal(err)
        }

        got := result.Content
        expected := "testtest"
        if !reflect.DeepEqual(got, expected) {
                t.Errorf("Test %s failed: got %+#v,
                expected %+#v", "LuaConn_Handle", got, expected)
        }
}
```

In the test, the connection is first retrieved from the lua scripter. After the connection has been successfully retrieved, a string function is set. This string function is later used in the handle method of the lua script. The string function returns test, and the lua function adds this string to the message it retrieves from the handle method in Go. After the strings have been concatenated, the concatenated string is returned to the handle method in Go. This is eventually tested against the expected string within the test. The separate functions like `GetConnection()`, `Handle()` and the retrieval of the result have been tested within another test.

5.5.2. Lua

While it is possible to add specific tests for lua, the goal of this project was to implement the possibility to add lua-scripts, not specific lua-scripts themselves. The Go-lua implementation was subject to human tests, before a choice was made which implementation would be used. So the connection between Go and Lua would be the best for all platforms. After the choice was made, the implementation of the vendor package was not tested further. Although the implementation itself was not tested, there was one case in which the build of the vendor package failed due to an update on the master branch of Go on Github. Due to this failed build an issue and pull request was opened on the Gopher-lua project. The fix for the issue was not merged by the administrator of the project within the time-frame of the project. So within the vendor package a small tweak was made to make the build work.

As it was not the goal to program lua-scripts, only methods that are made available to lua are tested. To do so, custom lua-scripts were made that use specific methods, the responses were then checked through default Go testing methods. These included methods contain functionality to retrieve information about the connection, log data to the console, download a file from an URL and send events to a channel. An example of such a test is the following:

```go
//TestGetLocalAddr tests the retrieval of the local
//address from a connection
func TestGetLocalAddr(t *testing.T) {
        got := getLocalAddr(connectionWrapper.Conn)()

        expected := "pipe"

        if !reflect.DeepEqual(got, expected) {
                t.Errorf("Test %s failed: got %+#v,
                expected %+#v", "getLocalAddr", got, expected)
        }
}
```

The `getLocalAddr()` method returns a function that retrieves the local address of an connection. This function is exposed to each script and can be used for general purposes, like checking if the connection is a new connection or an old one. The `connectionWrapper` is a testing structure which houses a dummy of a connection. Because the connection is a pipe dummy, it return the local address as `"pipe"`. This fact is used as testing purpose.

After the connection wrapper, the scripter connection and separate scripter methods are tested. Also the exposed functions, the exposure functions and the handle methods are tested within Go. It can be assumed that the working of the whole scripter flow works as well. Keeping the tests separate and not testing the flow in one test, ensures that additions or rewriting of parts does not influence other tests. Therefore it is easier to check which part of the software contains an unexpected bug or error.

5.5.3. Generic service

The generic service is the third part that was subject to tests. Testing the service implementation was more of a hassle. A service relies on far more dependencies than the scripter. Making a dummy for each and every dependency would be a major implementation and would require the dummy packaging of vendor packages. These vendor packages are, if it is right, tested by the makers. Therefore the testing can rely on these vendor packages as correctly working. The tests of the generic service can therefore focus primarily on the functionalities of the generic service.

The generic service has three components to be tested, the initialization, the handle of a connection and the methods that have been exposed to the Lua scripts, especially in the generic service. The general service is the hybrid way of programming a service, where the `CanHandle()` and `Handle()` method are being handled in Lua. The initialization phase is like any other service and does not contain any special logic. These two functionalities are therefore easily tested, but it must be taken into account that the Lua scripts need a lot of testing from the future developer. One small mistake can take the generic service completely down, or take one file down that can have a certain value in the swift world of cybercrime.

The last part of the generic service is the methods that are exposed to Lua. In the time of the project REST functionality has been build into the generic service. This means that HTTP requests can be read in and HTTP responses can be send back to the client. This functionality is available within Lua from Go methods that have been exposed. These methods are tested within Go, like the following example:

```
// TestHTTPRequest checks whether a HTTP request will
// respond accordingly
func TestHTTPRequest(t *testing.T) {
    req := httptest.NewRequest("POST", "/",
    strings.NewReader(`{"username":"test","password":"test"}`))
    if err := req.Write(client); err != nil {
        t.Error(err)
    }

    rdr := bufio.NewReader(client)

    resp, err := http.ReadResponse(rdr, req)
    if err != nil {
        t.Error(err)
    }

    body, _ := ioutil.ReadAll(resp.Body)

    got := map[string]interface{}{}
    if err := json.Unmarshal(body, &got); err != nil {
        t.Error(err)
    }

    expected := map[string]interface{}{}
    if err := json.Unmarshal([]byte("{\"login\": \"success\"}"),
        &expected); err != nil {
        t.Error(err)
    }

    if !reflect.DeepEqual(got, expected) {
        t.Errorf("Test %s failed: got %+#v,
        expected %+#v", "login", got, expected)
        return
    }
}
```

This HTTP request test is built upon a simple login functionality inside the `Handle()` method within a lua script. When a user tries to log in with a username "test" and a password "test", the lua script has to return a login success message. In the Lua script the HTTP request is read in and the fields are checked to be alright. If the fields are set and they are right, the login success message is written back to the response of the HTTP request. With this test several functionalities are tested in one flow, like the read in and the write of the HTTP protocol.

5.6. Miscellaneous

5.6.1. Remove unused connections

After running the newly added lua-scripts, the Honeytrap started filling up the memory. This was caused by not cleaning up new connections containing lua states for each IP-address. An interval was introduced that checks all stored connections after the elapsed time specified in the `config.toml`. This checks for connections that are not used for over an hour. If so, they are removed from the list of connections, allowing the garbage collector to clean them up and free the memory used.

5.7. Rethinking the services

After the first implementation of Lua-scripting was finished, some doubts formed about whether the chosen solution was the best way to integrate Lua scripting into Honeytrap. As previously described, there are two ways of using Lua scripting in Honeytrap in our initial implementation.

Firstly, there is the hybrid form, where the service is mostly implemented in Go. In this implementation in Go, some calls to the scripter are made.

The second implementation, which is used to implement a service entirely in Lua, has some small Go 'boilerplate', which serves as a connector from the service to the Lua scripter. In the scripter, everything on the incoming connection is handled, with the exception of some Go methods that are exposed to Lua. This full Lua service is implemented as a service, where iteration over all the scripts is done with the call to the `canHandle` method to check whether at least one script in the generic service can handle the connection. When the generic service receives the connection from Honeytrap it does an iteration again over all available scripts with a call to the `canHandle` to check which script can handle it exactly. This results in the service doing multiple calls to the `canHandle` method from the generic service in Go to the method in the Lua scripter. This hierarchy is somewhat illogical, calling the same method and passing through arguments to a different method which essentially does exactly the same.

A way to solve this is to make a component that sits between the `findService` and the implemented service. Currently, the director and Lua scripter are subcomponents of the service itself where the service implementation handles where the connection is handled. Design wise this is not the smartest way to do it, since the service-implementation, the director and Lua-scripter are components that handle the same thing. Placing them on the same level would be more beneficial for a more efficient implementation and allows for more flexibility in selecting the way the connection is handled.

Solving this can be done by introducing a connector, a component that allows switching between the service, director and scripter. This allows for switching dynamically between the components by using functions that are exposed to the service and scripter. The director won't be able to switch the connection to the other services since there is no control over the LXC container. Once the connection is in a container, the connection can only be changed to another location by having a point in between the connection from where the incoming connection is listened to before forwarding the messages to the container.

This connector will make it easier to handle services in Lua, since the Lua implementation is now quite bound to the service, making the Lua implementation feel like an afterthought. The connector allows for moving the handshake from the connection in the service implementation to the connector implementation. The handshake in the connector removes the need for difficult constructions in the service implementation, like switches to distinguish the handling modes for Lua, director and native Go implementation.

5.8. Documentation

There is a separate repository that creates a web-based documentation for Honeytrap. At the time of writing only a small amount of documentation is added to this repository and most pages only show: "Information about ... can be found here", but not more. For this project however extensive documentation was added, split over two different subject, configuration and implementation. The implementation tab had no subpages/menu yet, so the logic for this was added as well.

5.8.1. Configuration

One page was added that gives information about the different steps that can be taken to add configuration for the different scripter-functionalities, ie. the lua-scripter self, a generic scripter, extending a service.

5.8.2. Implementation

One scripter tab and four subpages were added that describe all the possibilities around the scripter:

1. **Adding scripter functionality:** A global explanation on the two ways of implementing a scripter are explained here, giving some minor information on the generic scripters and scripters that extend existing services.
2. **Extending a service:** This chapter explains how to extend a service with (lua-)scripting, first by explaining how to extend the gocode of the service itself and after that how the lua-scripts should look and where they should be placed.
3. **Generic:** This chapter explains how to implement new services/protocols in Lua.
4. **Methods:** This chapter explains what different methods can be called from lua to get specific information from Go, or perform certain tasks. These are split up for the global tasks and the methods that are only available in generic scripts.

6 Findings

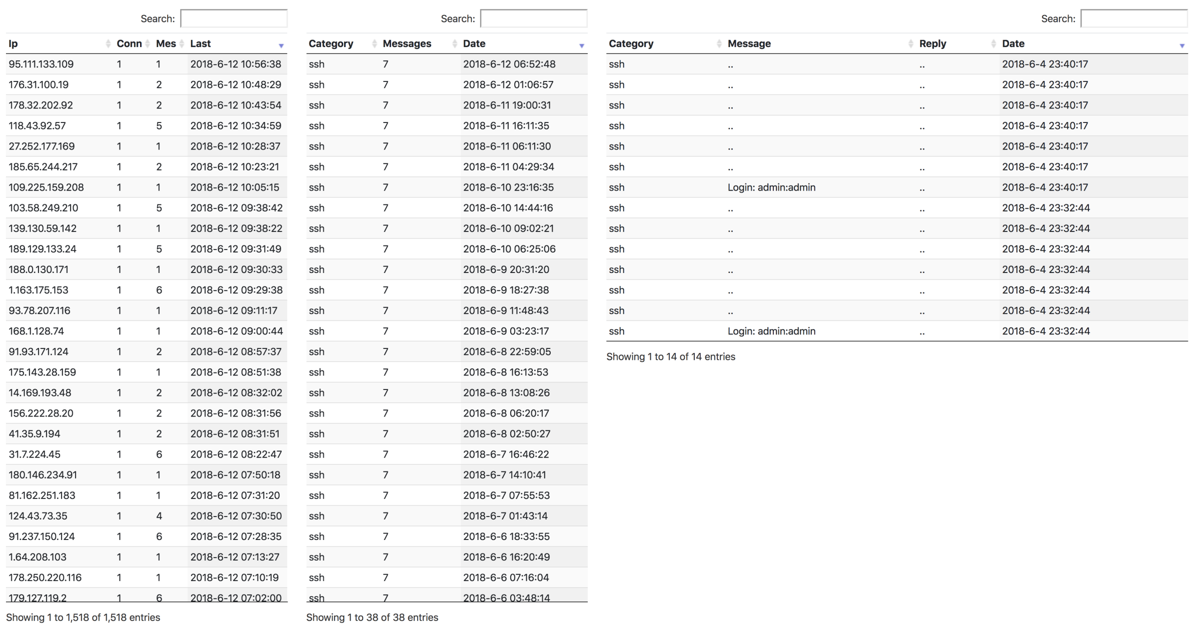
Different tools were available to discover the types of attacks that came in, tools like Elasticsearch and Kibana. Kibana showed a pretty visualization of Elasticsearch and showed a barred graphical view of the amount of attacks over a certain time frame. While this gave quite some insight on the type of attacks, a clear overview of what a specific hacker does was still absent. An extra tool was therefor created:

6.1. Custom stats tool

A custom statistics tool has been added that combines messages in Elasticsearch for IP-addresses and creates groups within an IP-address that are grouped by connection time, when a message is sent within 30 minutes of a previous one, it is added to that connectionGroup. The tool visualizes this by showing 3 tables.

1. The first being all IP-adresses with the amount of found connections and messages.
2. The second being all connections for a selected IP-address, showing the used protocols, the amount of messages and the last message per connection.
3. The third being all messages for a selected connection, showing the protocol, the message, the reply and the datetime.

This view allows users to quickly find new kinds of attacks that are performed and what calls are performed in combination.



Search:

Ip	Conn	Mes	Last
95.111.133.109	1	1	2018-6-12 10:56:38
176.31.100.19	1	2	2018-6-12 10:48:29
178.32.202.92	1	2	2018-6-12 10:43:54
118.43.92.57	1	5	2018-6-12 10:34:59
27.252.177.169	1	1	2018-6-12 10:28:37
185.65.244.217	1	2	2018-6-12 10:23:21
109.225.159.208	1	1	2018-6-12 10:05:15
103.58.249.210	1	5	2018-6-12 09:38:42
139.130.59.142	1	1	2018-6-12 09:38:22
189.129.133.24	1	5	2018-6-12 09:31:49
188.0.130.171	1	1	2018-6-12 09:30:33
1.163.175.153	1	6	2018-6-12 09:29:38
93.78.207.116	1	1	2018-6-12 09:11:17
168.1.128.74	1	1	2018-6-12 09:00:44
91.93.171.124	1	2	2018-6-12 08:57:37
175.143.28.159	1	1	2018-6-12 08:51:38
14.169.193.48	1	2	2018-6-12 08:32:02
156.222.28.20	1	2	2018-6-12 08:31:56
41.35.9.194	1	2	2018-6-12 08:31:51
31.7.224.45	1	6	2018-6-12 08:22:47
180.146.234.91	1	1	2018-6-12 07:50:18
81.162.251.183	1	1	2018-6-12 07:31:20
124.43.73.35	1	4	2018-6-12 07:30:50
91.237.150.124	1	6	2018-6-12 07:28:35
1.64.208.103	1	1	2018-6-12 07:13:27
178.250.220.116	1	1	2018-6-12 07:10:19
179.127.119.2	1	6	2018-6-12 07:02:00

Showing 1 to 1,518 of 1,518 entries

Search:

Category	Messages	Date
ssh	7	2018-6-12 06:52:48
ssh	7	2018-6-12 01:06:57
ssh	7	2018-6-11 19:00:31
ssh	7	2018-6-11 16:11:35
ssh	7	2018-6-11 06:11:30
ssh	7	2018-6-11 04:29:34
ssh	7	2018-6-10 23:16:35
ssh	7	2018-6-10 14:44:16
ssh	7	2018-6-10 09:02:21
ssh	7	2018-6-10 06:25:06
ssh	7	2018-6-9 20:31:20
ssh	7	2018-6-9 18:27:38
ssh	7	2018-6-9 11:48:43
ssh	7	2018-6-9 03:23:17
ssh	7	2018-6-8 22:59:05
ssh	7	2018-6-8 16:13:53
ssh	7	2018-6-8 13:08:26
ssh	7	2018-6-8 06:20:17
ssh	7	2018-6-8 02:50:27
ssh	7	2018-6-7 16:46:22
ssh	7	2018-6-7 14:10:41
ssh	7	2018-6-7 07:55:53
ssh	7	2018-6-7 01:43:14
ssh	7	2018-6-6 18:33:55
ssh	7	2018-6-6 16:20:49
ssh	7	2018-6-6 07:16:04
ssh	7	2018-6-6 03:48:14

Showing 1 to 38 of 38 entries

Search:

Category	Message	Reply	Date
ssh	..	..	2018-6-4 23:40:17
ssh	..	..	2018-6-4 23:40:17
ssh	..	..	2018-6-4 23:40:17
ssh	..	..	2018-6-4 23:40:17
ssh	..	..	2018-6-4 23:40:17
ssh	..	..	2018-6-4 23:40:17
ssh	Login: admin:admin	..	2018-6-4 23:40:17
ssh	..	..	2018-6-4 23:32:44
ssh	..	..	2018-6-4 23:32:44
ssh	..	..	2018-6-4 23:32:44
ssh	..	..	2018-6-4 23:32:44
ssh	..	..	2018-6-4 23:32:44
ssh	..	..	2018-6-4 23:32:44
ssh	Login: admin:admin	..	2018-6-4 23:32:44

Showing 1 to 14 of 14 entries

6.2. Kibana visualizations

Kibana can be used to make simple visualizations with the collected data in Elasticsearch. This data was used to create informatic graphs that can be found below.

6.2.1. Connection counts

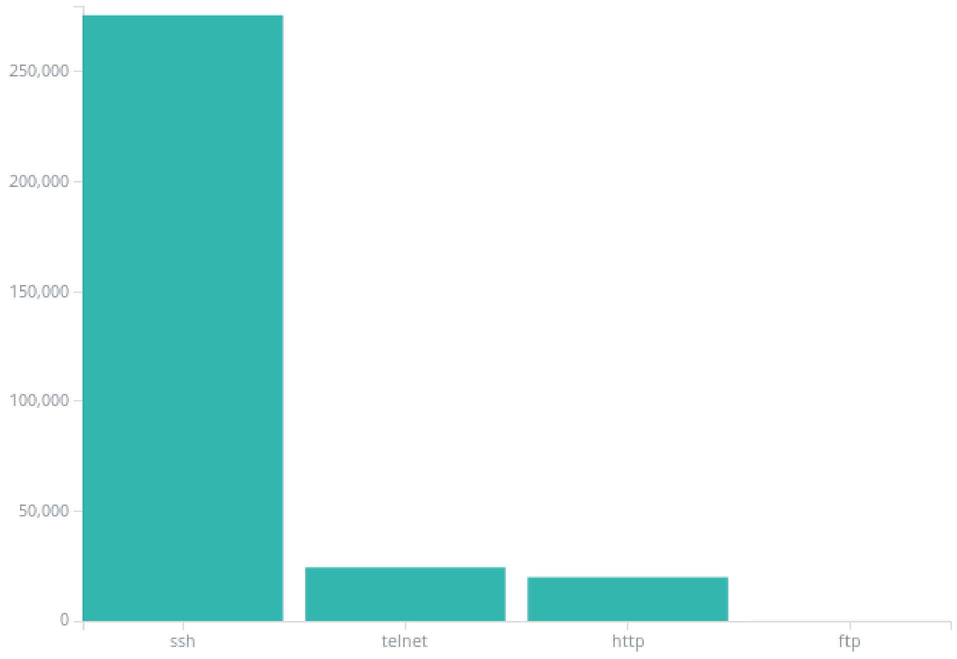


Figure 6.1: Requests to the services

The first graph shows the amount of separate calls that were made for each type of service. As can be seen in the graph above, SSH is the most used protocol by far. This is partly explained by the way in which SSH works. To be able to use SSH a user first needs to log in, which can be seen back in the amount of calls since many hackers tried hundreds of username/password-combinations before calling a real command. A very interesting observation is that there was a vast amount of login attempts where no command was sent. This looks like the attacker is doing some reconnaissance and collecting login credentials for the server, to be used at a later point in time.

Another graph was created that shows the same, but without login attempts:

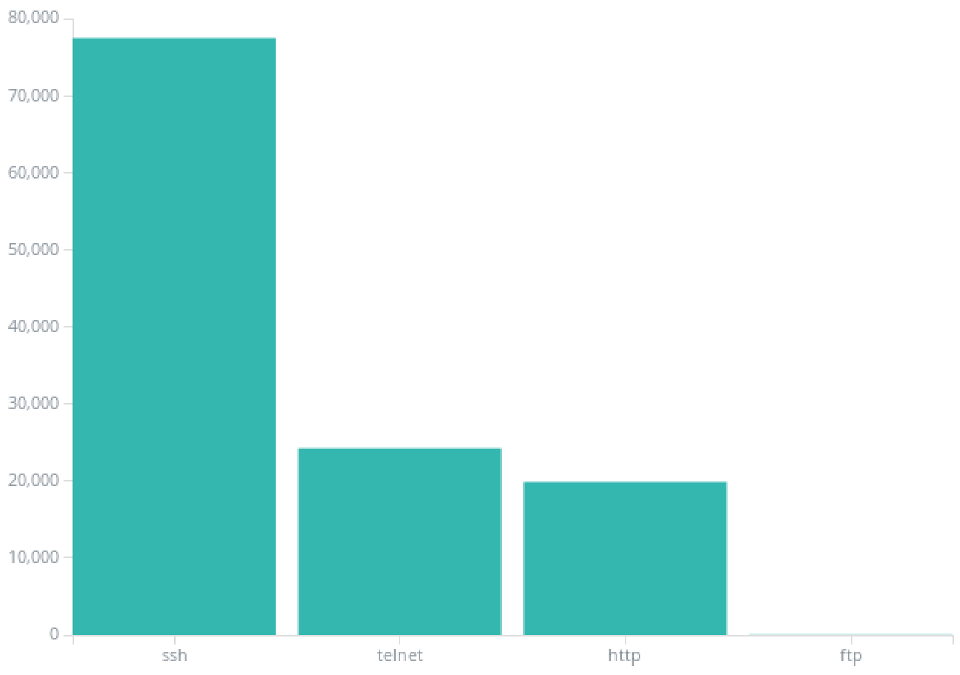


Figure 6.2: Requests to the services with SSH authentication requests removed

Even after removing login attempts from the statistics, it still seems that the SSH service is a lot more interesting for hackers to exploit.

6.2.2. Target IPs

The server instance of Honeytrap, which was run on a DigitalOcean droplet, had a couple of agents connected to it during the project. These agents forwarded each and every connection to the Honeytrap instance. An overview of the amount of incoming attacks per agent IP address for each service was made:

	SSH	HTTP	Telnet	Total
167.99.33.146	60,867	4,428	1,511	66,806
192.168.1.50	12,922	7,127	7,205	27,254
188.166.169.148	3,146	6,073	12,154	21,373
192.168.2.50	711	1,502	3,537	5,750
Total	77,646	19,130	24,407	121,183

A total of 3 agents were used during the project. As can be seen above 4 different ips give data, this because the two '192.168.*.*' IPs are positioned behind a router with the connections forwarded from the router, hence the source-ip being an internal IP. What is most interesting is that the 192.168.*.* ips were placed behind the domains airpparel.com/nl, but received a lot less attacks than the separate ip 167.99.33.146 which was placed at home. Giving the idea that having a random domain does not grant extra traffic/data.

6.2.3. Random domain-redirects

An interesting observation is that Kibana showed traffic that was redirected from other websites. An assumption is that these websites are hacked and compromised and want to redirect to malicious content on a server. It is very plausible that this is the result of an attacker thinking that he/she succesfully compromised the Honeytrap server and deployed malicious code on it.

6.2.4. Machine identification

The first few weeks most calls that were shown in Kibana were login attempts on SSH followed by the command 'uname -a' which asks the type of operating system is running. The Honeytrap did not have a correct answer for this at first which resulted in the 'Command not found' response, after which the hacker most often tried another login and again the 'uname -a'. After Lua-scripting was correctly implemented in combination with the AB-Tester a function was written that could return a random operating system based on the paramaters passed by the uname command (-a, -v, ..). While it was expected that follow-up commands would follow after returning an operating system, the opposite happened. After a day of incoming connections that received a correct 'uname -a' response almost all traffic through ssh stopped, resulting in a very quiet server. While this was the opposite of the goal, it is interesting to see that the Honeytrap server was most probably listed in some sort of database with the operating system that was returned. Since the hackers most probably have no exploits for this OS, the attacks stopped.

6.3. Found exploits

6.3.1. Dead malware links

In the logs of all the traffic received on the testing setup, various links and a collection of questionable content was found. Quite a bit of this content was invalid or expired, links were returning 404-pages or the entire server was down. This is one big flaw of a lot of malware, all the servers and URLs are hardcoded, meaning that if anything changes in the URLs or servers, the malware is producing incorrect results and has serious trouble functioning as intended.

6.3.2. GPON Exploit

On monday the 7th of May, a lot of requests were done over the HTTP protocol on the URL `/GponForm/diag_Form?images/` . While searching for this URL on the web a CVE came up that was released 5 days before. This CVE included a vulnerability in the GPON routers. CVE-2018-10561[5] and CVE-2018-10562[6] contain critical vulnerabilities where complete control over these types of devices and the network can be obtained through remote code execution. This vulnerability was used by the Muhstik botnet to accelerate expansion. Entering the command to download and execute the Muhstik malware. This idea was reinforced after other articles were posted containing the same information and confirming these suspicions.

6.3.3. Using found fake data

During a short period the SSH protocol was implemented via the first Go-lua implementation. During this period different automated scripts tried to login and retrieve, for example, the data structure. A file named `password.txt` was used as a lure, and a person logged in with ssh tried to read or download the `password.txt` file. On one occasion the `password.txt` file was acquired. After some time elapsed, the credentials were used to login to the SSH server using a completely different IP. The credentials were so specific that we can conclude that both sessions probably belong to the same person. This shows that there probably is a division between the "scouts", which find new machines to compromise and the "executers", which try to actually hack the system.

6.3.4. DDoS botnets

In the third week of the BEP the SSH Honeytrap server received and continuously executed commands with different URLs in it. The attackers downloaded the Perl script on the server, ran the script and deleted the original file. DDoS scripts are the most common, the different versions all look similar and seem to be based upon other scripts. They all use some form of security through obscurity, some changing every newline to 5 newlines, others obfuscating the script or even downloading the file by downloading another file first. The file that is downloaded initially is then executed and downloads the actual file. Usually, the exploit files provide a way to contact the creator with a skype account, a website or an IRC channel to which the hackers are all connected and issuing commands. These scripts often use an IRC channel as a command and control server, which have around 50 clients connected at a time.

6.3.5. Speedtest

Another interesting observation is the fact that one of the attackers started out with a speedtest, posting the results using the public speedtest API and printing the sharing link as output of the command. It can be the case that only the high-speed instances are filtered out for further exploitation, minimizing chance of getting caught but only using the more useful high-traffic instances.

7 Conclusions

7.1. Relevance of the software

While subjects like cryptocurrencies, AI and sustainability gain a lot more publicity since the last years, and give humanity many cool features, there is one other subject that might worry the most: News around DDoS attacks, hacked email-accounts and leaked data.

The internet is growing exponentially and more and more of our daily lives start to revolve around computers. While this makes our lives a lot easier, it also opens up many new ways for people to abuse it. Account data is being stolen, systems and websites are being hacked and complete services are put down by attacking it with thousands of hacked devices. It is quite hard to discover how hackers manage to hack into systems and steal data or even completely take over machines to use them for other hacking attempts.

This is where Honeytrap comes into place. Good hackers often hack into a system, get the data they want and then delete all back traces to them and information on how the hack was performed. A system that pretends to be a vulnerable device and logs everything the hacker does is therefor a good solution for the problem at hand. While the hacker thinks he is removing all data on how the hack was performed, all of this data is logged and can be processed by security companies to analyze and learn new hacking methods.

7.2. Proof of usefulness

While Honeytrap, before this project was started, already provided the possibility to log these hacking attempts, it didn't provide a way to swiftly react on new unknown types of attacks. When hackers discover a new exploit, they will start using it. Because the Honeytrap server is not a real server, the hacker will connect to it, try out the exploit, see that it doesn't work (since Honeytrap doesn't react correct on it), and continue.

The scripter of this project provides the possibility to see this new type of incoming attack and in a matter of minutes or hours, depending on the complexity, write a response that the hacker might expect, so that the user can see what the follow-up steps of this new type of hack/attack are. Providing the user with insightful information on the new types of hacks and allowing them to create fixes faster.

During the development of the scripter, a new exploit was discovered in GPON routers, after which the Honeytrap server was hit a lot by attackers that checked whether the server was a GPON router. Since Honeytrap didn't know what to respond, the attackers didn't continue with their hack-attempts. A Lua script was therefor written and later on the day hackers that connected thought they were indeed connecting to a GPON router and started redirecting the server to URLs that download their real virus binaries.

After also implementing a file download method that would download the files that hackers wanted the server to download, the first real virus binaries were found that could be reported to virus databases.

7.3. Proof of the impact

During the process of developing the scripting module within Honeytrap, the significance of the implementation was brought to the surface. Several hackers became aware of positive responses from our servers through automated requests. What followed were manual hacks and followups with new automated hacks based on the responses. Examples of these; Americans typing manual commands to our SSH server, calling scripts that were taught to be downloaded on the Honeytrap server after success response, retrieving IRC DDoS calls to activate new scripts and the forwarding of different domain names to the Honeytrap server that they though was compromised. All of these cases did not happen before the implementation of the scripting module went live, or would have happened without the positive responses by the Lua scripts.

This is clearly proof of the impact on the gathering of data within Honeytrap. Taking the steps to gather data further than only sensor mode or proxy a request to a container like LXC or a webserver. Lua scripting is a perfect fitting step between sensor and proxy mode. A quick response, also based on the history of commands, can be programmed to trick hackers into delivering more data from their hacks into a Honeytrap system. Without the chance that a hacker can really hack the system to use it for DDoS attacks or other hacks, which sometimes happens within LXC containers. The additional data that is gathered can be of significant impact on the company value for their clients. More data means more information and more information means more valuable actions can be taken to defy attackers.

7.4. Conclusion on the problem analysis

As stated in the problem analysis the main goal of this project, is that the user can respond in a very fast way on new types of exploits that are spotted in the Honeytrap. The scripter that was implemented in combination with the AB-tester provides the solution for this problem. A user can spot a new exploit, see on what port and with which payload an attacker is connecting and add this port to the server configuration, write a Lua script with a handle method that fetches the payload and give a reply that the attacker would expect to see. Or, when multiple replies could be given, use the AB-tester to give these different replies. Within a short time the user can add a first reply to a new type of attack and see what attackers would do next. Without having to rebuild the entire Honeytrap.

8 Discussion and recommendations

8.1. Retrospect

The project consisted of a few parts: Implementing a basic scripter, implementing a generic sripter, adding lua scripts to test the use of the new features and the start on the rewrite of the base of Honeytrap.

In retrospect the way of first adding support to existing services and after that support for new services in general was a good approach. Was it known earlier however that there were plans for the rewrite of the core of Honeytrap to support better switching between services and the scripters, it might have been a better idea to first do this rewrite and after that build the scripters. This because the implemented generic scripter and features for existing services will need some rewrites to work correctly for the new core.

Furthermore some work was done on the implementation of a web interface that could be used by users to manage the scripts in a more convenient way than placing them in the scripts folder manually. The logic for this was completely built into the Go project of Honeytrap and quite far into the web-interface of Honeytrap. After a meeting with Remco form DutchSec he then pointed out that while this was a good idea during the research phase, development had started on Honeyfarm, the enterprise edition of Honeytrap in which this feature should come instead of the web-interface itself. Making the work on it redundant. In retrospect this work should not have been done, since it would not be used at all.

8.2. Using the scripter

The scripter has been added, new services can be created and existing services can be extended. We recommend to write many of the new services and exploits that are planned for Honeytrap to be built in Lua or another to be added scripter language instead of Go itself. Implementing these elements in Lua gives other users an easier way to change how their specific Honeytrap reacts for each service and creates a less complex main project since the specific logic for all services and exploits is managed separately.

8.3. Extending the scripter

8.3.1. Adding more general lua methods

A few basic methods have been created that can be included by Lua to make it easier for users to create new scripts. These methods give the user encode/decode methods for JSON, a method to find a URL in a message or a method that lets a user send a personalized message based on a standard message. Since the idea of the scripter is that many users can create scripts in an easy way, we recommend that all methods that can be reused in later scripts, are separated and placed in the 'utils' directory so that others can include these in the future.

8.3.2. More scripter languages

The current scripter only provides Lua support. The way that it is built makes it very easy to add new languages. There are a few basic interfaces in scripter.go that need to be implemented and then a new language can be used as a scripter instead of Lua.

8.4. Honeyfarm

Near the end of the project Remco Verhoef of DutchSec pointed out that he was working on Honeyfarm, an enterprise edition for Honeytrap that would make it possible to manage all owned Honeytraps in a convenient way. The features added by the scripter and AB-tester can use some form of managing as well, we recommend to add an overview for both the scripter and the AB-tester:

8.4.1. Scripter

All of the current scripts are placed in a public GitHub repository which can be placed manually or through a submodule into the users' Honeytrap project. It would however be a lot more user friendly if it is possible to manage the active scripts through a web interface. This overview should connect to the GitHub repository and read out all files that are present. Each Lua script contains a specific list in the comments before the real script begins in which data is given about the last version and dependencies. This data can be used by the web-interface to see if there are new versions available for already existing scripts in the Honeytrap. This way the user can update these scripts, but also add new ones or delete existing ones without knowing anything about Git, Lua or the Go code.

8.4.2. AB-Tester

Besides the scripts, an AB-Tester was added in this project. These are now stored in the BadgerDB storage that is already used by Honeytrap. There is at this time no way of managing these values, except by editing a JSON file that sets all the key-value pairs. When you wish to change, add, delete AB-Tests you now have to update the file and restart the server, which is inconvenient. An overview in Honeyfarm that lets the user maintain all the different AB-Tests would be a good addition to the AB-Tests in general. Including the possibility to even create some statistics around these tests: how often have they been called, how often has a follow-up call be performed after returning an AB-test.

8.5. Connector

The connector rework is not completed, but it has been very useful to find the direction Honeytrap has to take in the future. The agility of Honeytrap increases greatly when introducing this component. The work that has been done for this components is a proof of concept, which shows that it should be included in Honeytrap in the future. This increases the extensibility of Honeytrap, which will increase maintainability by reducing the amount of responsibility the service components have. By splitting into smaller parts and a more logical responsibility distribution, this allows for easier onboarding for new contributors and overall better code quality.

The part for the connector that needs some more thought is how data from the handshake is passed to each of the handling modes, "service", "director" and "scripter". The easiest way would be returning an `interface{}`, which can be of any type. This allows for an FTP handshake method to return an `FTPStruct` which contains all information about the FTP connection that was initiated. This `interface{}` can then be cast to a `FTPStruct` in the different handling modes, making the connection data available to these methods.

8.6. Testing

Tests were not widely present in the main Honeytrap project. The coverage does not tell anything about the correctness of the tests, but test coverage was around 10% initially. During development tests were written for each method that was added to the Honeytrap in favor of the scripting module. Therefore the test coverage at the end was around 78% for the code that was added during this project. Which in fact does not say anything about the correctness, but the tests were written based on the principles of software quality and testing. A simple recommendation is to implement far more testing in the main Honeytrap project, to really emphasis the correctness of the software, also in edge cases. A single example is the port range bug that prevented a Ripple example on port 51235 from working, because of a byte error was. This was then fixed and for correctness added through a pull request to Honeytrap with three tests to prevent this happening in the future.

8.7. Comments

Comments are useful when they are written correctly. As per quote from one of the team members:

Comments are as good as you make them.

Looking at the Honeytrap project, a few important functions are commented. The comments are in our insight good, but could be improved by adding more comments to the internal flow methods. Especially to the initialization functions of Honeytrap in the `honeytrap.go` file. Which will add some more clarity to newcomers and provide useful information when debugging the software. Within the implementation of the scripting, effort has been made to make useful comments for defining methods and flows.

8.8. Documentation

The docs.honeytrap.io website hosts the documentation on the Honeytrap project. For the initial setup of our Honeytrap the documentation was used, but it did not provide enough information. While the setup was completed successfully one needs to look into the code base to get a grasp of how the project needs to be ran and which options are available. The documentation on the docs.honeytrap.io website is therefore more of a generic documentation on what is available within Honeytrap, not a software documentation on how to implement the service. With the scripting implementation a more in depth documentation is given on how to use the scripting functionality. The recommendation is therefore to give more documentation on the implementation of the main Honeytrap project for software engineers or system admins, and for companies to write more documentation on the advantages of the scripting module.

8.9. AB-Tester

While the AB-Tester that was implemented for this project adds some nice new features, there are some points on which it can be made a lot better. The main idea that has come up with during the project was to not create different replies for a certain commands, but create whole sets of replies that made sense in combination, an a way making a set of replies that form a 'machine'. When a user connects now and calls a command a random answer is given, when this same user performs another command, another random answer is given which could be compliant with the first answer. It would be nice if sets are created in which the answers for many different commands are set. Instead of getting a random answer for each call, get a random set for each user and fetch the command from that set, each time the user calls it.

9 Discussion on ethical implications

Gathering data about intruders and their sophisticated attacks, that is the goal of the project Honeytrap. Cracking a system without the permission of the systems owner and using it in your own advantage is commonly described as unethical. In most countries the act of cracking a system is a crime. This means that Honeytrap collects data from mostly criminals trying to intrude systems that are not their own. That statement sounds very ethical, but has some sharp edges which will be written out in this section.

The first implication of a sharp edge is the collection of unwanted or unnecessary data. Honeytrap provides the best results when the system acts like the official service the instance tries to replicate. This means that non-hackers can be tricked into using the system for real use, where in reality, it is a Honeytrap. The data it collects is from non-hackers that are visiting the system without the intent of hacking the system. That makes for unwanted or unnecessary data, because it can not be used in analyzing real threats and can even be a cloak for real hackers.

To make a Honeytrap most effective, the internal and external structure of a company or system must be known. If a bank wants to collect data on attackers and protect a service with Honeytrap, the service must be known to the company that provides the Honeytrap. This brings a vast ethical responsibility in which a trust relationship between contractor and supplier is a benefit. The exchange of information and data needs therefor to be formalized and happen in such a way that attackers do not have the opportunity to gather the information and data.

At last sharing of the gathered information on exploits and hackers can be of value to the whole computer science community as discussed in the problem analysis. But what type of information can be shared and what type of information can not be shared. Depending on each country there are already rules about sharing information on potential exploits and hackers, but these hacks often go across borders. Strict regulation is therefore needed between countries and organizations in the world, on what will be shared and on which level it will be shared. Sometimes it will be effective to share the information about an exploit in public, but on the other hand the public sharing of information on a potential hacker can be very ineffective. If a hacker is available on a public cybercrime wanted list he/she will be more careful about his/her actions, which will work in favor of the hackers.

A
Project Description

At DutchSec, a product has been created called Honeytrap. Honeytrap is a software honeypot to trap malicious hackers by slowing down and identifying their sessions when they enter your system. The service consists of a vast collection of ways to let external threats connect, log malicious activity and alert when necessary. We are currently looking to expand our honeytrap with support for lua scripting.

Currently services are build in the language Go. This happens to take a lot of time compiling and testing when new specific exploits need to be programmed. Lua is a cross-platform scripting language that is designed for embedded use in applications. When Lua is implemented into Honeytrap, we can start using Go-Lua implementations, at which the service is implemented within Lua instead of Go. This would mean far less compilation time, and it would be easier to implement specific exploits. This project focuses on the implementation and testing of the specific exploits in a Go-Lua implementation.

Other information:
What features, implementation and testing is needed to make handling of specific exploits less bound to deployments?

EXTENDING HONEYTRAP WITH LUA SCRIPTING

Presentation 04-07-2018

DUTCH SEC

ENVIRONMENT

The Project took place at DutchSec, a company of about one year old that specializes in software tools that gather, process and analyse data around security. Their main goal is to provide security for everyone: 'Because everyone should be able to use technology safely.'

CHALLENGE

Honeytrap, a software honeypot, is one of the products that DutchSec builds. A honeypot simulates server interaction and collects data. Honeytrap is unlike other Honeypots, instead of focusing on a single service, multiple services can be configured. This configuration makes Honeytrap very usable for different applications and services. The goal of the software is to make it easy for everybody to gather data on potential intrusions and take appropriate action against attackers. More and more private data is stored in the cloud, making it more interesting to search and exploit vulnerabilities in software. Speed is key in the collection of data on these exploits. Honeytrap did not provide the swiftness and easy going solution to react quick on new exploits. Providing a swift solution for this problem formed a great challenge in this project.

PRODUCT

During the project the Honeytrap system was extended with Lua scripting. Allowing users of Honeytrap to implement new services or extend existing services without having to change the code of Honeytrap itself.

Users can now create or download Lua scripts, place them in the scripts folder of Honeytrap and reload the scripts through the web interface by pressing a button. Without any restart or rebuild, the new service is now loaded and new types of intrusions can be found.

RESEARCH

During the research phase, information was gathered on other Honeypots and different ways of implementing Lua in Go. After which a benchmark was performed and finally the best Lua version was chosen.

PROCESS

The first weeks were used for implementing a basic version of Lua after which more functionality and exposed methods were added. Some extra visualizations and user-friendly functions were added in the end. At last a rework was initiated, which is still in progress and can be added later to the final product. The way of working was according to the Agile methodology since this closely resembled the way of working at DutchSec.

OUTLOOK

The pull request to the main product will be opened for final approval. User convenient functionalities should be added in the future, in Honeytrap or through Honeyfarm. Furthermore a 'work in progress'-pull request has been opened which rewrites the core to support better switching between services and scripts.

PROJECT TEAM

M.G.A. Janssen

Interests

Hacking, Security, Exploits and Go

Contributions

Generic scripter
Infrastructure
Connector
Malware analysis

N. van Nes

Interests

Analysing data, User behaviour

Contributions

Lua scripts
Benchmarks
Testing
Exploit management

T. Oomens

Interests

Teamwork, React, Security, Data visualization

Contributions

ConnectionWrapper
AB-Tester
Lua scripts (Ripple)
Stats Tool

Contributions by all

Report
Presentation
Code design
Code implementation

CONTACT INFO

Client: DutchSec
R. Verhoef
CHO, Co-Founder

TU Coach
Dr. C. Doerr
Intelligen Systems & Cyber Security

Contacts

M.G.A. Janssen	m.g.a.janssen@student.tudelft.nl
N. van Nes	n.vannes@student.tudelft.nl
T. Oomens	t.oomens@student.tudelft.nl

The final report for this project can be found at: http://repository.tudelft.nl

C
Research

The research report is based on the first two weeks into the project. In these two weeks the project was specified and defined based on the request from the customer and the research that was performed. After defining the path that had to be taken to deliver a successful application, research was conducted on the already existing product Honeytrap. At last the best way of implementing Lua inside Honeytrap was examined. In the research appendix these elements will be written out.

C.1. Project definition:

The first two weeks of the project were designated for doing research on all there is to know and find on the implementation of Lua in the Honeytrap project. This research was split into a few parts: How does Honeytrap work, what exactly does the client want, what options are there available for implementing Lua in Go, how exactly does Go work and are there similar projects to be found online. Honeytrap is a project that is open source, but in the early stages of development. Tests, comments and documentation were barely to be found, making it an extra challenge to understand the exact way that Honeytrap worked. The concept and setup of Honeytrap was very clear. Some research and experimentation before the BEP project, made Honeytrap something to comprehend quickly. The days after that different websites on implementing lua were searched for and discussed with the client. Also some searching was done for other Honeypots that were similar, all of which below a more detailed explanation is given.

C.2. Implementing Lua in Go:

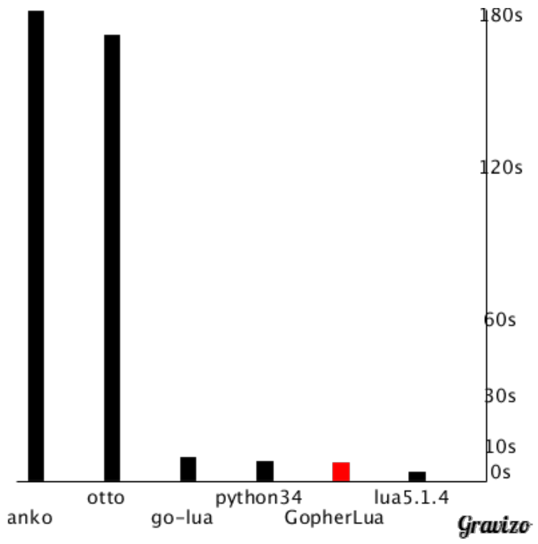
An implementation of a Go-Lua framework will be required to have Lua working on the HoneyTrap server. There were 4 options available, one being the option to create a Lua interpreter in Go from scratch and the other 3 to use open-source options available online. Below these options will be discussed and a conclusion on is given on which option is most favourable. The (o/c) indicates whether the issue or pull request is open or closed.

C.2.1. The different options:

Name	Commits	Last update	Issues (o/c)	Pull req (o/c)	Tests	
Gopher	232	16-04-2018	14/103	4/52	Lua 5.1 tar	Gopher-lua[12]
Aarzilli	122	02-03-2018	9/37	1/19	Minimal testing	Aarzilli[10]
Shopify	214	04-12-2017	26/22	4/43	Lua 5.2 tar	Shopify[7]
Custom	-	-	-	-		

C.2.2. Testing method:

Each of the implementations was tested using the following methods: Description of the method that is used to test the specific implementations go lua Performance, compile Lua to go, speed of resulting go code. Some Go Lua implementations use the test suite provided by the Lua team. This so called "test suite" contains a vast collection of different testing methods to ensure that Lua gives the correct results.



Gopher:
Lua version: 5.1
Go version: >= 1.8

Performance: In the wiki of Gopher some benchmarks can be found in which it is compared to other types of languages like JavaScript and other implementations that implement Lua with Go.

program	time
anko	182.73s
otto	173.32s
go-lua	8.13s
Python3.4	5.84s
GopherLua	5.40s
lua5.1.4	1.71s

As can be seen in these statistics there are some alternatives for the Lua language. These alternatives however are significantly slower than Lua, and are therefore no good replacement at this time. There are 3 other comparisons made, one with the Lua implementation of Shopify, one with Python3 and one with native Lua. As can be seen here, Gopher is slightly faster than Shopify and Python3, but still about 3 times slower than native Lua. It can be seen however that Gopher is quite a fast implementation of the Lua language in Go. As stated by the programmers: "GopherLua is not fast but not too slow, I think."

Logging: Gopher implements Loguago (https://github.com/rucuriousyet/loguago) for the logging of content, adding functions that allow for easy logging.

Activity: The repository for Gopher was created on the 15th of February in 2015 and has been highly maintained since then. Over 232 commits have been added since then, evenly distributed over the last 3 years. There are quite some open issues, yet also a large amount of closed issues, some dating back to a few days in the past. The conclusion here is that this repository is still actively maintained and has been so for the past 3 years.

Documentation: Gopher has an extensive documentation and wiki on how to use and implement it. All used libraries are mentioned and 90% of the code is documented and tested.

C.2.3. Aarzilli:

Lua version: 5.1, has beta's for 5.2 and 5.3
Go version: Unknown

Performance: GoLua does not list any performance ratings or comparisons. But since this implementation only consists of go bindings to the underlying C Lua interpreter, we can assume that it is significantly faster than the other methods. In the Go-Lua documentation, in the next section, it is mentioned that the Go-Lua implementation is about 6x slower than the C Lua implementation. From this, we assume that this implementation will be faster than the others. For a more complete view how this implementation compares to the the other methods, a benchmark is included as a later section.

Logging: GoLua does not do any logging. The only logging the framework does is when a exception has occurred and this is in the form of a Go panic. Which basically crashes the whole Go program.

Activity: This implementation uses the C Lua implementation, meaning that the Lua VM implementation itself is most up-to-date with the latest version of Lua. As long as GoLua implements the C Lua api correctly, the results are exactly the same. The activity for this repository is expected to be lower, only the C Lua API bindings have to be maintained. Still, this repository is active and has recent pull request merges and commits

Documentation: Documentation is sparse for GoLua, only edge cases are documented to avoid strange behavior and resolve issues. While this is fair, some more elaborate examples would be nice, since the current ones are not very well documented with comments or some other form of explanation.

C.2.4. Shopify:

Lua version: 5.2
Go version: >= 1.4.2

Performance: Shopify has tested their Go-Lua implementation against the C Lua interpreter and Gopher-Lua (mentioned above). The performance tests has been done with the Fibonacci function, starting with the first 35 elements. The Go-Lua VM from Shopify is around 6x slower than the C Lua interpreter. On the other hand, Gopher-Lua is around 20% faster than Go-Lua. Following the results:

Time (s)	Lua	Gopher-Lua	Go-Lua
Real	2.807	14.528	17.411
User	2.795	14.513	17.514
System	0.006	0.031	1.287

Logging: Shopify does not provide real logging, if a called function crashes it calls the panic function from Go. Which in fact crashes the whole program. Syntax errors are on the other hand captured before the run of the function.

Activity: The Go-Lua VM from Shopify has been used in Shopify's load generation tool since May 2014. It has been highly maintained from within the company, but not on issues reported outside of the company. Over 210 commits have been deployed, but almost all commits were at the beginning of the project in 2014. Pull request are mostly approved with an imbalance of 4 open versus 43 closed. The repository has been tested exhaustively via the provided Lua tests by the Lua organization.

Documentation: The Shopify Go-Lua framework has an extensive documentation on godoc.org. Almost all functions and variables have been documented and explained. Especially the required functions that are needed to build a good implementation in Go. On Github could be more explanation, because that is often the entry point of the project for newcomers.

C.2.5. Custom:

After discussing with DutchSec, the decision was made to make use of an already existing Lua implementation. This because, while adding Lua support is important, the focus of this project will be mainly about the custom functions that will be added to Lua to add a good support between Lua and Honeytrap itself.

C.2.6. Benchmarks:

To provide an insight in the workings of the different Go-lua implementations and the corresponding performance, a benchmark test was conducted. In the benchmark different factors are being evaluated, like initialization time, running time, ease of use, and implementation of functions in both directions (Go -> Lua and Lua -> Go). For the benchmark a lua-benchmark project was initiated where the three different Go-lua frameworks are implemented via Go. After making the frameworks work in the project, which was quite easy with the examples, a Fibonacci and prime function was implemented to acquire the first results. For the Fibonacci function, the first 1000 fibonacci numbers are calculated 100000 times. This is due to limitations with the integer sizes. For the primes, we calculate all prime numbers till 100000. We have also added one last test, combining the the two tasks and running them in the order Native -> Gopher -> Shopify -> Shopify -> Gopher -> Native. This will try to eliminate the initial startup time and possible cpu throttling. The results are shown below, with times in milliseconds:

	Native	Gopher	Shopify
Fibonacci	51	114	145
Primes	1797	9889	7184
Combined	21507	41858	49132

C.2.7. Conclusion:

The options can be split into three different groups: A custom implementation, a Go-implementation of Lua or a implementation through bindings to the original Lua.

The first option has been discarded, since the effort that needs to be taken to write a new implementation is not worth the time. While the implementation through bindings is the fastest, it has the downside that it is written in C, making it hard to be run on Windows. The speedup of the bindings is about 5 times compared to the Go-implementation, but because this latter one still has the speed of Python3.4, this loss in speed is negligible. Making the Go-implementations the best option at this point. With this in mind the option remains to implement both, making it configurable in Honeytrap to choose which implementation one wishes to use. Other features are however more interesting, so this will be low on the optional features list.

The last decision lies on which one of the two Go-implementations to use. The favour however goes to the Gopher-implementation, since it is slightly faster, better documented and has a more active repository than the Shopify-implementation. When implementing a first deployment of the Go-Lua implementation, the Shopify-implementation responded bad on date functions. The implementation could not handle the date and time functions from the OS. Gopher-Lua instead, did not had any problems handling these date and time functions. The final choice has therefore come to the use of Gopher-Lua.

C.3. Possible features:

While the first goal of this project will be to implement Lua into Honeytrap, the next steps will be to add many features that will make Lua extremely useful within the Honeytrap application itself. Below many different features are described that can be implemented and are then divided into groups according to the MoSCoW principle.

1. **Custom native Lua methods:**
   The main focus of this project is that users can build on the spot scripts to implement certain exploits within Honeytrap. While this will be possible to do with basic lua implementations, some things can be quite hard to implement each time through Lua alone.
   A good addition could therefore be to create Lua methods that add basic functionalities and can be used by each written script, eg methods that return the user id or global occuring data in each connection. These methods should make Lua-scripts that are created later easier to write since they will use basic methods that are given by the framework itself.

2. **An external documentation on all features added by Lua:**
   Since extra methods and functionalities will be added by the addition of Lua an extensive documentation will be required for end-users so that they can easily look up all the possibilities and the correct way

of implementing these added features. This documentation should be available externally, for instance via the documentation repository of Honeytrap itself.

3. **Honeytrap methods that are callable from Lua:**
   While methods that add specific features should be added, it could also be a good feature to add the possibility to call specific Honeytrap methods directly without the requirement to access them through the custom made lua methods. The benefits of this is that all (newly added) methods in Honeytrap will always be available even if the Lua methods aren't updated yet, allowing an end-user to always use the newest version of Honeytrap.

4. **Plugin-like structure for adding lua-scripts:**
   It would be a good feature to be able to add lua scripts in a 'plugin' matter, having a certain directory in which all these 'plugins' can be placed after which they will run their code when required too. This way one could write new implementations of lua-scripts and share them with others, which will only need to add these new methods to the 'plugin' directory.

5. **Deployment of lua-scripts over multiple Honeytrap instances:**
   Some users will have multiple Honeytrap instances running. When a lua-script is written it would be a good feature to be able to deploy additions and changes automatically to all Honeytrap instances owned by the user. An interface to handle this deployment should be created and a git like system to manage updates.

6. **Web implementation for managing/writing lua-scripts:**
   There could be multiple ways of managing/creating/editing lua-scripts, one of the methods could be a special web-interface that shows which lua-scripts have been added, which version of them and whether they are active or not. An addition to the current React web-interface should be created that shows a user which lua-scripts are installed at that time with a possibility to install more scripts.

7. **Separate website for sharing/downloading lua-scripts:**
   Ultimately a separate website on which something like a marketplace can be setup would be very useful to have. A website where one could upload all the created scripts, either the ones from DutchSec or the ones from others. With the possibility for users to download/install these scripts from others or even with the possibility to buy these. This will be quite a large system since it will require logins/ checks/ a vote system/ a backend for managing your scripts/etc. But might as well be the ultimate end goal of this project since it will allow all users of Honeytrap to be able to share hotfixes in a fast and effective way.

8. **Component-like structure for lua-scripts:**
   The Lua scripts should be able to be used in other Lua scripts. Enabling the user to build a specific (sub)method or function in Lua and being able to call this method/script in another Lua script.

9. **More to be brainstormed...**

C.4. MoSCoW:

MSCW	Feature	Time required	Depends on
Must have	Native lua methods Honeytrap methods		
Should have	Documentation Plugin-like structure		
Could have	Auto deployment Component-like structure Web implementation		
Would have	Separate website		

C.5. Honeytrap methods that have to be exposed in Lua scripts:

At first basic communication should be available for alteration. Each service should therefore include Lua commands at 4 points:

- The constructor, initializing the Lua scripts that are meant for that type of service.
- The start of the handle, intercepting the incoming message and altering it if wanted.
- In the middle of the handle where the resulting message can be overwritten.
- At the end of the handle, intercepting the outgoing message and altering it if wanted.

These additions to each service should make sure that Lua can perform all tasks it wants to do, concerning communication with an incoming client.
While these are the essential implementations some other methods that are defined in Honeytrap should be available in the Lua scripts:

- The logging methods: Lua should be able to write to the correct log at any time
- Time methods: All methods surrounding time should be available
- BadgerDB: The scripts should be able to connect and write/read from the database,

C.6. Testing:

It will be important to extensively test all added features. The working of Lua and all of its functions need to work correctly, but also all added methods and features that will be made within this project need to be tested. For the Lua side a specific test set is made available by the developers of Lua themselves, they have provided many scripts that can be run and of which the results can then be tested. All other features that are added will have to be tested by therefor created test cases, it will therefore be important that extra time is kept apart for the development of these tests.

C.7. Web framework:

The current framework that is deployed by Honeytrap is built with React. The resulting website gives some basic information about sessions, but not that much more since Kibana can already give quite a big visualization. When a web implementation for Lua scripts is added this will have to be done in React as well, to keep a similar working approach. Initially an extra tab could be added to the current generated website, showing all installed scripts and adding the options to add/delete scripts.

C.8. Description of workflow:

Honeytrap is an open source software package. This means that all code is already out in the open, therefore the newly written code will be too. A Github organization is made which houses the different repositories cq. forks for the project. The first fork is of the main Honeytrap project in which the Go-Lua implementation will be built. The second repository is for the benchmark of the different frameworks. The workflow of adding new features and fixes bugs is as follows; make a new branch that clearly describes the purpose of the branch, code the needed solution for the feature or bug, push in the desired branch, make a pull request, at least 1 person needs to approve the change and Travis needs to successfully build the project. After this, the branch can be merged. If the branch can not be automatically merged, a rebase will need to be done by one of the programmers. For the pull request, a squashed merge will happen, in this way each commit on the master branch will have a clear description of the underlying feature or fix, without showing all commits the original PR consists of.

C.8.1. How to actually implement Lua scripting in Honeytrap

Running Lua within Go is something that will be quite easy to do. There are some elements that need to be taken into account, in which problems might occur which are discussed below, followed by the possible solutions.

- Do we want to script everything after the initial connection to the service is established, or do we only want to react on specific events/occurrences?
- Do we want multiple Lua scripts interacting with the same connection?
- Can another Lua script take over the connection at some point from other scripts?
- Or is every script listening to the connection and does it take over at some point, prohibiting the rest from taking ownership?
- Multiple exploits can be used by attackers, on which different scripts react. Do Lua scripts handle prioritization themselves or does Honeytrap implement this feature?

After talking with Remco from DutchSec, where the above questions were asked, some decisions could be made: The vision is to have the Lua implemented as a service. The service reacts on the exploit when it recognizes the communication. The service acts stand-alone from the protocol. In this way, several Lua scripts can act on the same protocol, or on the same command. The owner of the Honeytrap is responsible for concurrent scripts on the same exploit. The first round of exploits that will be covered are zero-day exploits, to react on these zero-day exploits, the scripts act on simple "if this than that"-cases.
Example: zero-day exploit where a password can be copied from a server, attackers try to download the password file. A Lua service is created to react on this operation, and is deployed quickly to a Honeytrap server. Attacker retrieves a 'fake' unique password from the Lua service. The next step of the attacker can be followed by, for example, the unique password. The step that follows the 'hack' of the password can be written in any other protocol by a Lua service.

C.8.2. CanHandle:

This method in Honeytrap checks whether the incoming connection can be handled by the service that the port is bound to. Eventually, the canHandle method will not be bound to a port, but listen to all incoming connections. The result of this is that every service can connect to every port, increasing the amount of accepted connections on the Honeytrap. By doing this, the best handle method will be used to handle the incoming connection.

C.8.3. Handle:

The handle method in Honeytrap takes care of really diving into an exploit. It handles actions, event and commands that come from the attacker. The function reacts with different actions like, logging, writing to file, giving a response and backtracking the user. Like the input is unlimited, the responses can be unlimited. To react quickly on new exploits is would be very useful to be able to program inside a Lua script. That Lua script does not to be compiled again before running it as a service on the Honeytrap server. This provides hugh advantages above writing the counters for the exploits in Go. Which needs to be build and deployed again.

C.8.4. History:

Another aspect that would be beneficial for the scripts is storing all connection information, including all previously sent and received messages on the connection. The canHandle and handle method will base their response on the history built with the connection. The history can be saved by IP, port or classifier.

C.8.5. One Lua script after the connection handshake:

The easiest method to allow Lua scripting is forwarding connection data after the handshake to the Lua script. Only one Lua script is allowed to interact with the connection, making this relatively easy to build and maintain. The Lua script can implement a specific interface or call methods from Honeytrap that have been exposed to Lua, allowing them to be called there.

C.8.6. Multiple scripts handling a connection with prioritization:

The chance exists that multiple scripts want to react on a certain incoming call. This can be in the form of read-only or by taking over the connection completely. A dilemma occurs here since a choice needs to be made to which script the connection will be sent. This can be done by prioritizing the scripts and allowing the one with the highest priority to interact with the connection.

C.8.7. Multiple scripts handling a connection:

The other option to handle having multiple scripts interact with one connection is leaving control to the user. Trying to find a correct prioritization solution and taking into account the history is a difficult thing to do. So by leaving control to the user, this removes the issue.

C.8.8. How to call Lua scripts from Honeytrap itself:

There should be a general agreement on how to write the basis of each Lua-script, this so that Honeytrap knows how to initially call the methods in each Lua-script. Each script should have a method that registers to Honeytrap when it wants to be updated on incoming data, for instance incoming connections. Furthermore there should be some required methods in each Lua-script that can always be called from within Honeytrap. Think of basic methods like: init, event-listeners or disconnect.

C.9. Example exploit:

An HTTP exploit tries to put unwanted files on your server and run them calling a URL. The goal of such an exploit is often to get access to restricted areas or to perform DDOS attacks. Uploading of unwanted files happens through upload fields on the website or via FTP on the server. To register the exploits and provide an environment to attract hacker, these two services need to be implemented. A known exploits is SUC027 : Muieblackcat setup.php Web Scanner/Robot[4]. The server that was online during our research phase got a lot of request on the /muieblackcat url. A list of known urls that are used for the exploit are:

- "GET /xampp/phpmyadmin/scripts/setup.php HTTP/1.1"
- "GET /websql/scripts/setup.php HTTP/1.1"
- "GET /web/scripts/setup.php HTTP/1.1"
- "GET /web/phpMyAdmin/scripts/setup.php HTTP/1.1"
- "GET /typo3/phpmyadmin/scripts/setup.php HTTP/1.1"
- "GET /scripts/setup.php HTTP/1.1"
- "GET /pma/scripts/setup.php HTTP/1.1"
- "GET /phpmyadmin2/scripts/setup.php HTTP/1.1"
- "GET /phpmyadmin1/scripts/setup.php HTTP/1.1"
- "GET /phpmyadmin/scripts/setup.php HTTP/1.1"
- "GET /phpadmin/scripts/setup.php HTTP/1.1"
- "GET /phpMyAdmin/scripts/setup.php HTTP/1.1"
- "GET /phpMyAdmin-2/scripts/setup.php HTTP/1.1"
- "GET /phpMyAdmin-2.5.5/index.php HTTP/1.1"
- "GET /phpMyAdmin-2.5.5-pl1/index.php HTTP/1.1"
- "GET /php-my-admin/scripts/setup.php HTTP/1.1"
- "GET /mysqladmin/scripts/setup.php HTTP/1.1"
- "GET /mysql/scripts/setup.php HTTP/1.1"
- "GET /myadmin/scripts/setup.php HTTP/1.1"
- "GET /dbadmin/scripts/setup.php HTTP/1.1"
- "GET /db/scripts/setup.php HTTP/1.1"
- "GET /admin/scripts/setup.php HTTP/1.1"

- "GET /admin/pma/scripts/setup.php HTTP/1.1"
- "GET /admin/phpmyadmin/scripts/setup.php HTTP/1.1"
- "GET /MyAdmin/scripts/setup.php HTTP/1.1"
- "GET /muieblackcat HTTP/1.1"
- "GET /wp-login.php HTTP/1.1"
- "GET /blog/wp-login.php HTTP/1.1"
- "GET /wordpress/wp-login.php HTTP/1.1"
- "GET /wp/wp-login.php HTTP/1.1"

To handle this exploit the following steps are implemented. First the attackers need to get a positive response on the URL, thus believing that a service like PHPMyAdmin, WordPress or XAMPP is available on the server. The attacker will then check if the server is vulnerable for an exploit. Currently the next step is not available known in the research. The response to the following step is not provided in the research. Guessing the attackers will try to login on WordPress or PHPMyAdmin a positive response need to be implemented and let the attackers try to do their thing.

WordPress is the most used CMS on the entire web. Around 30%[8] of all websites are build via WordPress. This is very interesting for attackers, if they find a vulnerability the exploit can be used on many websites with confidential information. There are a lot of articles on the web that describe these exploits in WordPress and how to prevent them Attacking WordPress[1], using Content Injection Vulnerabilities in WordPress[2] and Unauthorized Password Reset in WordPress[9]. With the Go-Lua implementation a faster response to these exploits can be accomplished and a fix can be made way quicker. The speed-up benefits include: more data, more information on the steps within the exploit and logging of these exploits to different services. With these benefits, for example, unusual behaviour can be detected and logged to the WordPress website. So the developers immediately have access to the information that is needed to fix the vulnerability.

C.9.1. Which functions need to be available from Go in Lua:

Lua needs data and information from the main system Honeytrap to correctly respond on a new connection or command. There are two ways this could be accomplished. First put everything on the stack of Lua before running the script. The second is to register functions on the stack of Lua and put them as a key-value pair in the main table. The first requires less programming work for the Go-Lua implementation, but in the end this implementation requires far more programming work for the Lua scripts. The second option requires more programming work in the Go-Lua implementation, but in the end the Lua scripts will be more readable and do not need much data parsing and manipulation. Lua scripting will therefore be faster when more functions will be available from Go and this is the goal of the Go-Lua implementation. In this project the second option is seen as the best option and basic functions will be implemented in Go to be used in Lua. These basic functions will include information about the connection, like IP, Port, Protocol and history of command, and information about the system like time, type of OS and location. This information is standard in providing responses to new connections and commands.

D
SIG feedback 1

After two weeks of research and four weeks of implementing the desired product, feedback on the code implementation was given by the Software Improvement Group, also known as SIG. For the first of two feedback rounds, SIG used an automated test to determine the maintainability of the written software. The original dutch feedback of the first round was the following:

> De code van het systeem scoort 3.0 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Complexity en Unit Size vanwege de lagere deelscores als mogelijke verbeterpunten.
>
> Voor de voorbeelden hebben we gekeken naar de bestanden die volgen het diff-bestand ook daadwerkelijk door jullie zijn aangepast.
>
> Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld, zijn aparte stukken functionaliteit te vinden welke gerefactored kunnen worden naar aparte methodes.
>
> Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een descriptieve naam kan elk van de onderdelen apart getest worden en wordt de overall flow van de methode makkelijker te begrijpen.
>
> In dit project is de oorzaak van beide metrieken vaak dezelfde: methodes bevatten te veel verantwoordelijkheden, waardoor de hoeveelheid code al snel vrij lang en complex wordt. De meeste van die methodes waren al in het systeem aanwezig, maar jullie hebben er nog een paar toegevoegd, waarvan SetBasicMethods() in methods.go met afstand de meeste complexiteit bevat. Over het algemeen zien jullie "nieuwe" methodes er goed uit, dus probeer dezelfde aanpak ook op dit soort uitschieters toe te passen.
>
> De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid testcode ziet er ook goed uit, hopelijk lukt het om naast toevoegen van nieuwe productiecode ook nieuwe tests te blijven schrijven.
>
> Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

Constructive feedback was given on Unit Complexity and Unit Size. These two factors are a direct implication of the size of the written functions. An example was mentioned in the feedback, one of our function 'SetBasicMethods()' too big and therefore did not fulfill one purpose, but several inner functions and logic. Splitting these functions into logic parts would make our code better maintainable. A second important part of the feedback considered the testing of the software. As the main Honeytrap project did not contain a lot of tests, the tests that were written were alright. Following these tests, new tests needed to be added for the newly written software.

Bibliography

[1] Attacking wordpress | hackertarget.com. https://hackertarget.com/attacking-wordpress/. (Accessed on 06/22/2018).

[2] Content injection vulnerability in wordpress 4.7 and 4.7.1. https://blog.sucuri.net/2017/02/content-injection-vulnerability-wordpress-rest-api.html. (Accessed on 06/22/2018).

[3] dutchcoders/marija: Data exploration and visualisation for elasticsearch and splunk. https://github.com/dutchcoders/marija. (Accessed on 06/25/2018).

[4] Muieblackcat setup.php web scanner/robot. https://eromang.zataz.com/2011/08/14/suc027-muieblackcat-setup-php-web-scanner-robot/. (Accessed on 06/22/2018).

[5] Nvd - cve-2018-10561. https://nvd.nist.gov/vuln/detail/CVE-2018-10561, . (Accessed on 06/25/2018).

[6] Nvd - cve-2018-10562. https://nvd.nist.gov/vuln/detail/CVE-2018-10562, . (Accessed on 06/25/2018).

[7] Shopify/go-lua: A lua vm in go. https://github.com/Shopify/go-lua. (Accessed on 05/31/2018).

[8] Wordpress stats: Your ultimate list of wordpress statistics (data, studies, facts - even the little-known). https://www.codeinwp.com/blog/wordpress-statistics/, . (Accessed on 06/22/2018).

[9] Wordpress core <= 4.7.4 potential unauthorized password reset (0day). https://exploitbox.io/vuln/WordPress-Exploit-4-7-Unauth-Password-Reset-0day-CVE-2017-8295.html, . (Accessed on 06/22/2018).

[10] aarzilli/golua: Go bindings for lua c api - in progress. https://github.com/aarzilli/golua. (Accessed on 05/31/2018).

[11] paralax/awesome-honeypots: an awesome list of honeypot resources. https://github.com/paralax/awesome-honeypots. (Accessed on 06/22/2018).

[12] yuin/gopher-lua: Gopherlua: Vm and compiler for lua in go. https://github.com/yuin/gopher-lua. (Accessed on 05/31/2018).

[13] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. Iotpot: Analysing the rise of iot compromises. In 9th USENIX Workshop on Offensive Technologies (WOOT 15), Washington, D.C., 2015. USENIX Association. URL https://www.usenix.org/conference/woot15/workshop-program/presentation/pa.