Department of Informatics

Master Thesis

Non-destructive verification of blind injection vulnerabilities

Erik Vetle Larsen
Supervisor: Øyvind Ytrehus

June 16, 2020

Abstract

Injection vulnerabilities have plagued the computer security industry for ages, and despite significant efforts made in both technical defenses and developer awareness, they continue to persist. Blind injections are a sub-class of injection vulnerabilities that are especially tricky to verify and exploit from a computer security testers perspective, as they inherently yield very little information about the targets internal structure even when successfully exploited. This thesis will investigate the history of injection vulnerabilities, culminating in blind injections, and then outline and develop a prototype for a program made to assist computer security professionals in verifying blind injection vulnerabilities in their daily work.

Acknowledgements

I would like to thank my supervisor Øyvind Ytrehus for the help, support and guidance he has provided throughout my degree, and for his patience with my horrendous study methods and all-nighter approach to getting things done. I have probably not been an easy student to guide, but your efforts have been highly appreciated. I would also like to thank my employer, Netsecurity, for both giving me inspiration for the subject of this thesis, and for providing me with a great environment to bounce ideas and questions off.

My time at the Department of Informatics has been very educational, and I highly appreciate that I have been allowed to walk my own path, doing project courses on my own fields of interest when possible. I would like to thank my fellow students, especially the other masters students at Simula, who have been been a great support during both my masters thesis and courses, and for being a great crowd to unwind with during less productive times. Both the academic and social environment has been a massive support in my growth as a person, and as a informatics professional.

Contents

List of Figures

Listings

Chapter 1

Preface

1.1 Commonly used concepts

Hacker	An expert at programming and solving problems with a computer [1]. Often used to describe people talented within the field of computer security.
Penetration test	Penetration testing (or pen testing) is a security exercise where a cyber-security expert attempts to find and exploit vulnerabilities in a computer system [2]
Vulnerability	A computer vulnerability is a cybersecurity term that refers to a defect in a system that can leave it open to attack. This vulnerability could also refer to any type of weakness present in a computer itself, in a set of procedures, or in anything that allows information security to be exposed to a threat. [3]
Exploit	Exploitation is the next step in an attacker's playbook after finding a vulnerability. Exploits are the means through which a vulnerability can be leveraged for malicious activity by hackers; these include pieces of software, sequences of commands, or even open-source exploit kits. [4]
Payload	In the context of a cyber-attack, a payload is the component of the attack which causes harm to the victim. [5]
Daemon	A daemon is a type of program on Unix-like operating systems that runs unobtrusively in the background, rather than under the direct control of a user, waiting to be activated by the occurance of a specific event or condition. [6]

1.2 Commonly used abbreviations and protocols

PoC	Proof of Concept
HTTP/HTTPS	Hypertext Transfer Protocol and Hypertext Transfer Protocol Secure
DNS	Domain Name System
LDAP	Lightweight Directory Access Protocol
GUI	Graphical User Interface

Chapter 2

Introduction

Blind injection vulnerabilities are a class of vulnerabilities in computer systems where an attacker is able to supply crafted input that is interpreted as code, but the computer system gives no indication of whether the attack was successful or not. Verifying instances of these vulnerabilities often rely on permanently changing the behaviour of the computer systems, but in penetration tests of live equipment you want to avoid harming or changing the behaviour of the equipment. This presents a challenge, as it is easy to simply erase data or shut down the computer system to prove that code was executed, but this harms the owner of the computer system, so how do we prove that we achieved code execution without destroying or otherwise harming the affected computer system?

2.1 Motivation

An important part of a penetration test is providing proof and a example case of the vulnerability found, commonly known as a proof of concept, but this can conflict with the wish to impact a company's operations as little as possible. The overall goal of a penetration test is to find and validate security vulnerabilities, and advice on how to improve them, thus it is counter-productive if the act of finding and verifying vulnerabilities causes downtime or other loss of productivity. Depending on the system and organization involved, a payload which harms the system can cause everything from a minor inconvenience to major system downtime. Major system downtime in business critical systems can equate to significant monetary loss for the affected company, thus it is any penetration testers goal to strike a balance between providing quality PoC's for vulnerabilities, while also not posing any unnecessary risk of damage to the systems they are hired to help improve.

For injection vulnerabilities these concerns are especially important, as an exploitable vulnerability can potentially allow the tester to run arbitrary system commands completely unchecked. The threat of an arbitrary code execution vulnerability is obvious, but verifying these vulnerabilities can be a risky endeavor even with benign payloads. Some systems are built and configured to do one singular task, and may act erratically if forced to do something it is not built for, which is exactly what an exploit is doing. These risks have forced penetration testers and white-hat hackers to develop novel methods to create PoC's.

One of these methods to detect if code has been executed without changing or impacting the computer system in question is to craft the payload in such a way that if it is successfully executed then the computer system will make a network connection to some other system specified by the attacker. Making use of this, the tester can make the system "call back" to some system they own themselves, and if they received a network connection they can consider the PoC complete and working. Having some predictable and consistent manner of registering and logging such callbacks across multiple protocols is highly useful in ensuring any potential blind vulnerabilities are verified in a non-destructive fashion.

2.2 Goal

The goal of this thesis is building a prototype system that can assist penetration testers in their efforts to verify blind injection vulnerabilities across multiple protocols and systems. Some inspiration is taken from the tool Burp Collaborator by Portswigger, which is a part of the penetration testing software suite Burpsuite by the same developers.

Burp Collaborator acts primarily on HTTP(S) and DNS, and generates a random subdomain under the burpsuite domain, and will log any interactions to the user. Building a system that expands on this by taking a modular approach will allow for injection vulnerability verification in systems like LDAP servers or other directly hosted services, which does not provide a web interface or any HTTP messages. Thus the foundational work will be building a system capable of logging and reporting on DNS and HTTP(S) queries. As these protocols are part of the backbone of the internet, these are extremely useful to make use of. As the plan is to make a modular tool that can be usable for more than just one singular purpose, we will also implement callback logging and reporting for LDAP, as LDAP is a protocol commonly used in organization internal networks, which provides a significantly different attack scenario than DNS and HTTP(S).

In addition to creating said system, a major goal is to gain a deeper understanding of injection vulnerabilities, more specifically the sub-class called blind injections, and the ways in which they can be verified, and what has been done previously by researchers and technicians in the field. The protocols selected for the initial prototype are also extremely important protocols which are a major part of how the internet, and internal networks, works today. For many developers and other high-level technicians these protocols are abstracted away by tools and other interfaces, but having a solid knowledge of how these protocols work and interact on a lower level is extremely beneficial, and often necessary, when approaching systems with the intent to exploit them and otherwise abuse functionality for results unintended by the original developers.

2.3 Overview

In chapter 3, this thesis will provide some background on the history of injections attacks, and why they hold such significance in the digital threat landscape today. Furthermore it will delve into what injection attacks are from a technical perspective, what they consist of, and what differentiates blind injections from regular injections.

Chapter 4 will discuss the requirements and design process of building a system which will be able to provide the functionality required. Both the big picture architecture, and the individual protocols and systems involved will be addressed from a design perspective. It will answer why each part is required, what significance it holds, and elaborate on why each module is important.

Chapter 5 will dive into the technical details of actually developing and implementing the elements outlined in chapter 4, and provide insight into how they work. The steps taken to install, build, configure and make use of each module of the tool as a whole will be discussed, and it will explain how the modules are tied together into a cohesive system.

Chapter 6 will discuss how the process from initial design to complete implementation went, and elaborate on some major challenges and failures experienced while working on the technical implementation. How the system has turned out will be compared to the initial description and design, and any deviations will be looked into and explained.

Chapter 7 will conclude with how the results of this project ties into the initially presented challenge of providing high quality PoC's to blind injection vulnerabilities in a non-destructive manner. It will also conclude with the effectiveness of callback logging and registering as a vulnerability verification method, compared to other currently applicable methods of blind injection vulnerability verification.

Chapter 3

Background

3.1 History of injection vulnerabilities

Injection vulnerabilities are a ever-present challenge to security professionals and software developers everywhere today, but they are not a new form of vulnerability. Compared to many other presently applicable vulnerability classes, injection vulnerabilities are very old, and despite significant research into correcting them and defending against them, they continue to be as relevant as ever. Pinpointing exactly when and where injection vulnerabilities first arose can be a challenge as many of the original technical computer vulnerabilities were found by hackers and other technical personnel, and not academic researchers. This results in the original early documentation being somewhat lacking, and gives few pointers towards where to read the original documentation.

In addition to this, the classification of "injection vulnerabilities" had not yet been established in the early days of research into the field. Regardless, we can pinpoint one of the very first cases of documented SQL injection, which is one of the most common forms of injection vulnerability on the web [7]. In 1998 the hacker magazine "Phrack Magazine" published an article by the user rain.forest.puppy documenting a series of vulnerabilities, including a case of what has later been recognized as the first instance of SQL injection. In the article rain.forest.puppy, which is an online alias for Jeff Forristal [8], investigates several NT Web Technology vulnerabilities. SQL injections are described in detail in the chapter "ODBC and MS SQL server 6.5". In the chapter the flaw is described without its current name of SQL Injection, but merely as a vulnerability allowing execution of batch commands in MS SQL servers [9].

The development of security measures for SQL instances and similarly vulnerable services following the initial discovery of injection flaws can be summarized with the issue of how to securely handle user input. As seen in rain.forest.puppy's article, the flaw consists of user supplied input being placed directly into system queries, which can be abused by supplying command symbols to alter the function of the original statement. Thus the main measure against injection flaws is to "Assume all end-user input is hostile" [10].

Treating all end-user input as hostile sounds quite simple in theory, but can prove extremely challenging in practice.

The challenge from properly sanitizing all user input arises for multiple reasons. One of the simpler ones to understand is simply lack of knowledge on the side of developers. If a developer has no understanding for the risks involved in handling user input, then the systems they build will have a high chance of containing such vulnerabilities. Other reasons involve legacy systems which would be extremely tough and resource intensive to modernize and improve. Another notable issue is the sheer amount of variations on injection attacks, as some defenses might work against some types of injections, but be useless against other types.

When actually fixing the fundamental coding flaws allowing for SQL injections in the first place is considered too taxing, too resource intensive, or otherwise not practically feasible, other defensive coding techniques were applied. The most common method to attempt to "cover up" a injection based vulnerability is by hiding any error messages. While a non-defensive service will crash and dump the error log to the user if it's supplied with user input that renders it's internal code invalid, the defensive coding technique of not revealing internal information to users can be applied to masquerade these crashes and prevent any information about structure leaking to potential attackers. In it's simplest form, detecting a SQL injection flaw can be as basic as merely submitting a single quote (') into some input field, as single quotes are control characters in SQL syntax, and observing if the system crashes, dumps error stack traces or otherwise acts erratically. [11] Even if the system is vulnerable, there is the possibility that the developers have replaced all error feedback messages and stack traces with some generic error message, or no error message at all. We are then presented with the issue that is the focus of this thesis: blind injection vulnerabilities.

The rise of blind injection attacks came as a natural evolution as defensive techniques were developed to prevent information leakage, but the actual underlying vulnerability was not fixed. The steps taken in blind injection attacks can be summarized as performing the same kinds of attacks as in SQL injection, but instead of relying on error dumps to gain information about the database and its structure, attacks where designed to trigger conditional responses based on the control symbols used in the malicious input, or the syntactical correctness of the queries after the attacker input has been inserted. Having covered the history of both regular injections and how blind injections came to be, it is important to gain some foundational understanding of how these vulnerabilities and attacks works in practical scenarios.

3.2 Technical details and anatomy of injection attacks

To gain a better understanding of how blind injection attacks differs from regular injection attacks, it is helpful to first establish a solid understanding of injection attacks in general. As injection attacks can vary widely in how they appear, we will use SQL injections as the base example, as we have already established that it is one of the most common variants. It is important to note that even though syntax and protocol/system specifics will vary, most injection attacks follow the same structure and pattern as SQL injections. If SQL injections are understood, then that knowledge can be extrapolated to most other injection vulnerable systems given knowledge of the internal workings and language syntax of said other system. For SQL query and code examples in this chapter, control words will be color coded blue, while input and input variables will be color coded red.

SQL is a query language used to interact with databases, and is very common in websites and services that rely on databases to store their information. The language essentially consists of queries asking for selected entries from specified tables which match some condition. Open Web Application Security Project presents the following example query: [12]

SELECT id, firstname, lastname FROM authors

This query will retrieve the tables id, firstname and lastname from the table authors when given some conditional to match. An example application can be some website that serves information about authors, but needs their unique identifier (ID) for some purpose. In this case the user might be asked to supply the firstname and lastname values for the query to retrieve the id, and the query can be modified and expanded to:

SELECT id FROM authors WHERE firstname = 'some_input' AND lastname = 'other_input'

Here the user will supply the firstname and lastname to complete the search. If there is an entry in the database with matching firstname and lastname, then the query will return the entry's associated ID value. If no entry is found then the query will simply return that no matching value was found. An observant reader will notice that the input is color coded red to signify that it is controled by the user, while the single quotes around are part of the query itself so the user won't have to submit their data with single quotes.

As we have mentioned in the background chapter, these single quotes are control symbols and are used to signify the start and end of string values. So what happens if we submit a single quote as part of our query? If we for example submit firstname as `evil` and lastname as `'attacker`, then the actual query sent to the database will change to:

> SELECT id FROM authors WHERE firstname = 'evil' AND lastname = ''attacker'

Since the query reads two single quotes immediately after each other for lastname, the database will interpret the query as submitting an empty string for lastname and then try to execute `attacker'` as SQL syntax, which will result in a syntax error, and the query will fail. The flaw reveals itself here, since we were able to submit input in such a way that the service attempted to execute our input as part of the query. From this point we can tailor our input to craft syntactically correct SQL statements, which can contain malicious commands. There are multiple ways to insert malicious commands into a SQL query like this, but to show an intuitive example, we will look at a simple example of modifying the conditionals of the original query in a way which will change the result entirely. An example attack which will make the query return every entry in the author table can be achieved by submitting anything for firstname, and for last name `' OR '1'='1`. Submitting this will result in the full query becoming:

> SELECT id FROM authors WHERE firstname = 'anything' AND lastname = '' OR '1' = '1'

Take note of where the red color coding begins and ends, and how it interacts syntactically correctly with the rest of the full query. In SQL, as in boolean logic in general, AND takes precedence over OR, resulting in the last part of the query in boolean terms looking like `(firstname AND lastname) OR 1=1`. Following boolean logic, it does not matter if firstname and lastname matches anything, as 1 will always equal 1, and the OR operator will turn the statement into a tautology. Because of this the logical result will be true for every single entry in the table, resulting in the SQL query returning the requested fields for every entry in the specified table. It is easy to imagine how this outcome can be devastating when queries are interacting with tables containing sensitive information.

For a regular SQL injection attack, when the input contains malformed commands, the service usually responds with an error specifying what kind of issue occured. These errors often come in the form of a stack trace, depicting the code or query used to parse the faulty input. [12] Using these error messages, an attacker can gain information about the structure of the code they are attacking, and thus craft payloads tailored specifically for that service. In the case of SQL Blind Injection attacks, these error messages are instead replaced by generic error messages, or no message at all. [13] The task of crafting a working payload becomes much harder (but not impossible) when the service you are testing simply states that "something went wrong".

In order to verify a blind injection vulnerability, we have to find some methods to prove that a service is indeed vulnerable. As mentioned in the introduction, our goal is to prove the vulnerability without damaging the service running. Thus we can not simply run a command meant to delete the database or otherwise crash the running service. There are multiple methods and tools at our disposal, but we will look into two of the most common ones. In the case of SQL injections, or any other injections that send commands to a database, time-based attacks are often used. These kinds of attacks attempt to gain information about the execution of potential payloads by investigating the time the service spends executing the command. An example of a non-destructive way to verify a blind SQL injection in a time-based manner, is running commands that intentionally will take a long time to execute, like sleep commands. By doing this we can observe if the service returns results in the regular amount of time, or if it freezes and returns nothing for a much longer time. By applying time-based methods in our attacks, we can craft payloads that we will know have executed successfully if they run slowly.

Another non-destructive method of verifying blind injection vulnerabilities, and the one which will be the focus of this masters thesis, is crafting payloads which will cause the system to execute code that will establish a network connection. In it's most basic form, this can be an injection payload which will execute a ping command towards some attacker controlled system. By submitting such a payload, a penetration tester can verify if the vulnerability was successfully executed by paying attention to any connections back to their systems. The advantage of this method over, for example, creating a reverse shell is that the payload is much simpler to craft and does not contain any code snippets capable of harming the system. This helps a penetration tester in ensuring that their involvement is strictly logged, and any potential external issues can not be pinned to the ongoing test.

A practical example of this method being used can be found in a service contained within PortSwiggers BurpSuite, called Burp Collaborator. Burp Collaborator was created by PortSwigger to complement BurpSuite in detecting blind vulnerabilities. Collaborator is hosted as a single server, which hosts a DNS service, a HTTP/HTTPS service and a SMTP/SMTPS service. It acts as the authoritative DNS server for it's own domains, and uses a valid signed wildcard TLS certificate. With this setup Burp Collaborator can detect any DNS lookups made towards it's domain, and can log web and mail requests made.

Burp Collaborator works very well in tandem with the web application penetration testing capabilities that BurpSuite as a whole offers, but Burp Collaborator lacks the functionality to easily apply the same principles to stand-alone network services. It can of course detect domain interactions, but has no support for registering callbacks from, for example, vulnerable LDAP services. So how do we start developing a solution that can be extended to support arbitrary protocols and functionality?

Chapter 4

Technical details

For the creation of such a solution we can take inspiration from Burp Collaborator for the base functionality of DNS lookup registering, and for how to provide a listener that properly log interactions. To go beyond the capabilities of Burp Collaborator we need to implement a more generic structure to how we handle interactions. Creating a interface that will be able to properly register and respond to all possible protocols is infeasible, considering the arbitrary amounts of encoding or encryption that might be involved in some of the more specialized network protocols. To circumvent this, the solution needs to have a highly modular structure, to enable easy additions of support for more protocols.

4.1 Platform

All of the services mentioned in the previous chapter has to be running on a computer somewhere. Choosing how and where to run the service is an important choice, as this would dictate how accessible the service will be from arbitrary locations on the internet. Since the program has the goal of being able to test arbitrary protocols and systems, developing it in the cloud will make it easy to access while also ensuring a high degree of uptime, in contrast to running it on my own computers. For this reason I have chosen to rent a virtual machine from Digital Ocean's cloud environment. This is a efficient method that allows me to easily create and scale virtual resources on demand, while also being cheap. The virtual machine will also be a Linux Ubuntu 18.04.3 (LTS) x64 machine, since this is a recent and stable version of a well suited Linux distribution.

Digital Ocean also provides easy access to helpful features like performance statistics, networking configuration, private networking and more. Having the ability to easily change both aspects of the VM, and the network it is placed in, is a big advantage over hosting the system on-premise.

4.2 Domain Name Server configuration

Logging and responding to DNS queries means we have to set up our own private DNS resolver which will reply and log according to our own needs. For this I have selected the Internet Systems Consortium's open source Bind9 name server. The reason for selecting this specific DNS implementation is that Bind9 is both open-source and under active development, while also being produced by ICS which is a recognised and trusted vendor for DNS and DHCP networking technologies.

An important aspect of the DNS requirements is the ability to create new subdomains on-demand. Bind9 supports this through a feature called dynamic DNS. Making direct use of dynamic DNS can be more complex than, for example, depending on the API of an ISP, although it ensures fewer dependencies. Fewer dependencies lowers the risk for any component introducing breaking changes, so the stability of the entire program will be increased.

4.3 Hypertext Transfer Protocol (Secure)

Setting up a webserver can be done in multiple way, but since this project merely needs a web service that responds to and logs requests without any requirements to serving any actual content, we can make use of pythons built in SimpleHTTPServer. This server is usually used for testing purposes, or for providing access to files in a directory. The logging capabilities are the most important feature for us, and combining SimpleHTTPServer with built in python logging tools allows us to direct all relevant output to a file, which allows us to easily verify any interactions made towards specified subdomains by querying said logs.

4.4 Lightweight Directory Access Protocol configuration

LDAP is a very common lightweight protocol for interacting with directory servers. This can mean and involve a multitude of things, but if we investigate the official website for the protocol, we can read that:

> "LDAP is the Lightweight Directory Access Protocol. It's a standards-based protocol that sits on top of TCP/IP and allows clients to perform a variety of operations in a directory server, including storing and retrieving data, searching for data matching a given set of criteria, authenticating clients, and more." [14]

A quote like this still might feel rather vague, but as is mentioned, LDAP is often used to interact with network shares and perform client authentication. From this description one might feel that LDAP sounds a lot like Active Directory (AD) in the Microsoft software ecosystem. The similarity comes from the fact that where LDAP in itself is just a protocol designating how computers can communicate, AD is a directory services implementation that uses LDAP to communicate, alongside another authentication system called Kerberos.

AD is extensively used in many companies considering it is a service from Microsoft, and wherever AD is in use there will be LDAP communication, which is why being able to register callbacks through LDAP queries is extremely useful. Verifying injection vulnerabilities that results in LDAP callbacks can make for a high-profile finding, as abusing the affected resources authentication level towards other directory servers in the environment can be extremely dangerous depending on what service is affected.

Implementing a LDAP service involves a lot of consideration and architectural design work because of it's inherent flexibility, but for the purposes of this thesis we merely need a running LDAP service that will log all inbound queries, and respond to them according to the protocol. What the response contains is not important, as we only need to log the queries for the given subdomain. Thus we can implement a rather barebones LDAP server, without having to spend too much time on setting up the hierarchies and other organizational structures needed by a proper directory server.

4.5 Modular Protocol Request logging

Ensuring that all requests over an arbitrary protocol are handled and logged properly, we will need modules that comply to the syntax of the specified protocol. In our case, where we will base our modular approach on the HTTP, DNS and LDAP protocols, we will require some method of receiving, parsing and replying to the given commands, while also logging them in a reliable and easy-to-parse way. An example of parsing and logging HTTP requests can be seen in the previously mentioned Burp Collaborator program.

Creating said modules will require different approaches depending on the protocols to be used. Despite this, the general architecture can be centralized with regards to logging. For the modules, we need to configure a set location to log interactions to, which we then configure in the overarching system that parses said logs. In this way we can access and process logs for arbitrary supported protocols through, in our case, python code. By creating python functions that parse the logs, we can merely call these functions from the web interface to gain access to the logs.

4.6 Web interface

We need some method of interacting with the system in order to obtain subdomains to register callbacks to, and to visualize the logs in an efficient and centralized manner. Since this thesis focuses on the callback registering and logging efforts, a simple framework has been chosen to create the Web GUI; Flask. Flask is based on python, and allows us to create most of the web server logic in python without having to include lots of different other frameworks and technologies.

By building simple templates based on HTML and CSS, and allowing for calling python functions directly, this is a efficient approach that allows us to focus efforts on the system architecture and logging procedures, instead of spending lots of efforts on the visualization. For the prototype, the actual web interface will not be a big focus, as the underlying protocols and callback registering functionality is the critical components. Allowing for flexibility in future development is considered good practice as it prevents having to severely rebuild significant parts of the program from scratch if any design changes are to be made.

The most minimalist interface needed would be some button or other interaction point to generate a subdomain the user can apply to their payloads, and some search field that will allow the user to input their subdomain and query for any callbacks in the logs. There are multiple ways to actually implement this with different degrees of flexibility and user accessibility concerns, but for this prototype a minimalist approach will be taken.

Chapter 5

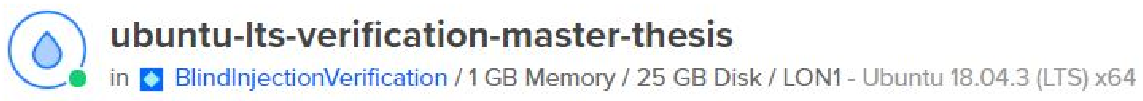
Implementation

5.1 Platform acquisition and set-up

Before we can start implementing any of the services I have outlined, we need to actually get a virtual machine to run things on. Achieving this was done by registering an account on digitalocean.com, submitting some payment method, and creating a project. Then we can freely create droplets, which are general-purpose virtual machines. From the "Create Droplet" view we can freely configure the resources and properties of the virtual machine, and I have selected a Ubuntu machine with 1 GB memory, a 25 GB disk, and some extra monitoring metrics enabled.

This is one of the cheapest configurations available, but also allows us to add more resources if the system eventually gets starved. After accepting the selected configuration, we are served an IP, and we receive an email giving us a one-time password to ssh into the droplet. After ssh'ing in, we finish the basic setup process by running `sudo apt-get update && sudo apt-get upgrade`, and this updates and upgrades all packages running on the droplet. Beyond these steps there's not much left to do in terms of preparing the virtual machine, as the image it is built from is set up with everything needed for basic functionality. Further installation and configuration will depend on the services and programs we install and use.

Figure 5.1: Cloud VM used for hosting the service



ubuntu-lts-verification-master-thesis

in BlindInjectionVerification / 1 GB Memory / 25 GB Disk / LON1 - Ubuntu 18.04.3 (LTS) x64

5.2 DNS setup

Installing the DNS service Bind9 on a Ubuntu machine is as simple as running `sudo apt-get install bind9`. This will download and install both Bind9 and any missing dependencies. Further we will configure the Bind9 instance so that it can be queried from anywhere, disallow zone transfers, and make the server authoritative-only. The default Bind9 configuration only allows queries from clients on the same LAN, and since we wish to be able to use the service from anywhere we will configure it to allow and respond to queries from anywhere on the net. Zone transfers are a feature in the DNS protocol where a client or another DNS server can request all DNS records for an entire zone, meaning that you can request all entries the DNS server has. This is considered bad practice to keep enabled as it gives away more information than necessary, and instead of allowing zone transfers by default you can set up exceptions for specific zones if necessary. Lastly, making the server authoritative-only means that it will only respond to queries for domains that it is explicitly configured for. This implies that the DNS server will not do recursive queries upstream when asked for domains it does not know itself.

We have to set up logging for the DNS service. For our purposes we need to log every query done towards the DNS, and which domains that were asked for, which is done by enabling the query_log channel in the Bind9 configurations file. By default Bind9 logs alle events to `/var/log/syslog`, which we wish to change for readability and ease of access. Thus we create a new folder `/var/log/named` where we then create four separate logging files for different types of events. For this thesis the query.log file will be used extensively to investigate if callbacks from pentesting activities are successful, but it is also helpful to log debug, security and system events to ease troubleshooting. These configurations are set through editing the `/etc/bind/named.conf.options` file. The contents of the full file can be seen in the Appendix, listing 7.1.

For the purposes of this thesis we will be using my own domain vetle.io as the hosting domain. This can easily be changed to whichever domain the service should be hosted at, since one merely needs to set the given domains name server record to point to the service. To start out we will create a zone file with some values that are required for the nameserver to be able to resolve queries correctly. The initial zone file for the `vetle.io` domain can bee seen in listing 5.1. The first line and subsequent indented lines specify that the current server is authoritative for the `vetle.io` domain. The two NS lines specify which servers act as nameservers for the domain, and we can see both the server itself (`ns.vetle.io`) and the Domeneshop slave nameserver (`ns.hyp.no`) being named. The following 4 A records are the IP address for Github Pages, which is where my own website is hosted. These four records hold no significance over this thesis, but serve as a convenient example of Bind9's ability to keep static entries alongside the more dynamic and short-lived subdomains used for callback registering.

Listing 5.1: /etc/bind/db.vetle.io

```
$ORIGIN .
$TTL 604800       ; 1 week
vetle.io                  IN SOA  ns.vetle.io. root.vetle.io. (
                                  24          ; serial
                                  604800      ; refresh (1 week)
                                  86400       ; retry (1 day)
                                  2419200     ; expire (4 weeks)
                                  604800      ; minimum (1 week)
                                  )
                          NS      ns.hyp.net.
                          NS      ns.vetle.io.
                          A       185.199.108.153
                          A       185.199.109.153
                          A       185.199.110.153
                          A       185.199.111.153
$ORIGIN vetle.io.
ns                        A       134.122.108.46
```

As we have mentioned earlier in the specifications of our DNS setup, we will rely on the dynamic update functionality of Bind9. Since this functionality will allow other programs than Bind9 itself to edit zone records, enabling this should only be done through the use of a secret key. There is no restriction on how this key should be generated, but it is recommended to create a secure one in accordance to best practices for its usage. For this project the tool dnssec-keygen has been used to generate a 256 bit key. The generated key is then placed in the /etc/bind/ folder. No naming convention is demanded, but we will name it `ddns.key` for simplicity, and the format the file has to be on can be seen in listing 5.2. The reason behind the syntax and variable name used for the key is that the Bind9 instance will be importing the key for verification of update queries.

Listing 5.2: /etc/bind/ddns.key

```
key DDNS_UPDATE {
        algorithm HMAC-MD5.SIG-ALG.REG.INT;
        secret "<secret_key>";
};
```

Now that we have a key saved, we need to actually configure the DNS zone to allow dynamic updates. By default this is turned off. The file named.conf should be left as it is, as it merely imports the other more specific configuration files. To create a DNS zone, and subsequently allow dynamic DNS updates, we have to edit the named.conf.local file to include our zone. The configuration of the domain vetle.io which will be used for this project can be seen in listing 5.3.

Listing 5.3: /etc/bind/named.conf.local

```
include "/etc/bind/ddns.key";

zone "vetle.io" {
        type master;
        notify yes;
        allow-update { key DDNS_UPDATE; };
        allow-transfer {194.63.248.53;};
        file "db.vetle.io";
};
```

The `allow-update` entry is what turns on dymanic updates, and by using the previously generated key as a parameter we configure Bind9 to require the key to perform any updates. The `allow-transfer` entry restricts DNS zone trasfers to the IP specified in the argument, which in this case is Domeneshop's slave DNS. Using a IPS slave DNS let's us focus on only one nameserver, as a domain requires two DNS servers in case of server outage or downtime. This allows the Domeneshop slave nameserver to request the zone file from our nameserver periodically, which ensures that if the primary nameserver somehow goes down, then the last recorded version of the zone file will still be operational. In the event that the primary nameserver is unavailable, we will be unable to make any new DNS updates, and some of the newer ones might not have been transferred to the slave nameserver yet, but any pre-existing records will still be resolvable. The IP addresses for my actual website on vetle.io is in our case a static record in the zone file, which I would prefer that stays resolvable even if this system goes offline for some time.

In theory, dynamic updates should be working now, but in practice there are still some obstacles remaining. When running Bind9 now and trying to perform a dynamic update, Bind9 will respond that it attempts to update the zone file, but then goes quiet without publishing any new records. Investigating the logs from the service reveals that while attempting to write to the zone file, the service runs into permission errors. While an extremely easy workaround to this would be to merely escalate the privileges of the DNS server, doing so would put our system at increased risk in case an attacker manages to exploit the Bind9 instance. One of the accepted workarounds to this is to create a bind folder in some writeable location, for example /var/cache, and then symlink the zone file in /etc/bind to the same zone file name in /var/cache. In our case we have symlinked /etc/bind/db.vetle.io to /var/cache/bind/db.vetle.io. By creating such a symlink, whenever Bind9 attempts to write to the zone file in /etc/bind it will be redirected directly to /var/cache/bind.

After overcoming the previous hurdle, the Bind9 instance is now configured for dynamic updates, and will create, edit and remove entries on demand. Interacting with the server is done through usual DNS commands, but writing and scripting DNS queries from scratch can be a hurdle. To overcome this, we make use of a python library called dnspython. This library gives access to tons of DNS query functionality through python functions. The web interface that we will look into later will be responsible for performing these queries, but to show a simple example of how dnspython performs it queries, I wrote a simple `update.py` script that will take a string value as argument, and create a new A record with the value of `127.0.0.1` in the zone, and then restart the nameserver to force the update to become visible immediately. The script can be seen in listing 5.3, with the key censored.

Listing 5.4: update.py

```
#!/usr/bin/env python
import os
import sys
import time
import dns.update
import dns.query
import dns.tsigkeyring

if(len(sys.argv) != 3):
        print "Error, wrong amount of arguments"
        sys.exit("Usage: python update.py <domain> <IP>")

keyring = dns.tsigkeyring.from_text({
    'DDNS_UPDATE.' : '<secret_key>'
})

zone = "vetle.io"
update = dns.update.Update(zone, keyring=keyring)
update.add(sys.argv[1], 8600, 'A', sys.argv[2])

response = dns.query.tcp(update, '127.0.0.1', timeout=10)
time.sleep(0.5)
os.system("sudo /etc/init.d/bind9 restart")
```

In addition to updating the zone file through DNS queries, it can be done by manually editing the zone file, although it is not recommended unless the DNS queries are not working or there are other special circumstances. When Bind9 is running with dynamic updates enabled, it will enforce a ”journal” of the updates. This involves keeping track of the update number both in its own interal systems and in the zone file. If the zone file is manually updated without correctly updating the journal number and subsequently loading the updated zone file in the correct manner, a ”journal out of sync” error will arise and the Bind9 service will crash. Resolving this error can be done by removing all journal files in /etc/bind and /var/cache/bind, and then restarting the Bind9 service instead, forcing Bind0 to regenerate journal files based upon the journal number found in the zone file.

Now that dynamic updates and DNS functionality has been established, we need to make verify that the logging we established earlier is working as intended. We specified that all query logs would be sent to `/var/log/named/query.log`, and retrieving callbacks from this file will be as simple as searching through the file for any entries with the domain we are looking for. If we have added the subdomain `test.vetle.io`, and some client queries that subdomain, then Bind9 will log the interaction on the format seen in listing 5.5.

Listing 5.5: Bind9 querylog entry example

```
14-Jun-2020 16:14:29.226 queries: info: client @0x7fc5e40c7540
<IP of client>#37448 (test.vetle.io): query: test.vetle.io IN A
-E(0)DC (<IP of nameserver>)
```

Searching through a log file can be done easily with python, and finding an interaction like this is proof that some network connection was made. If the subdomain was created recently for the purpose of some test, it is not likely that any crawlers will have found it yet, and even less so if the subdomain is a random string of characters. The validity of the callback can be double-checked by matching the IP in the log entry against the IP of the system that was exploited.

5.3 Hypertext Transfer Protocol (Secure)

Setting up a simple HTTP server is quite a simple task as there exists plenty of tutorials online, for all degrees of complexity. For our purposes, we only need a web service that responds with some arbitrary content and logs the interaction, and most importantly the host header used in the HTTP request. The host header is the domain or IP used to access the server, and this is where the subdomain created for the callback will appear if successful. Pythons built in SimpleHTTPServer can be called as merely a one liner, but this would expose all the content in the current working directory to the internet. To avoid this, we will write a python program to define the behaviour when receiving GET requests. Since the content of the response doesn't really matter, we will make the HTTP service return the string "200 OK" upon receiving a GET request. Alongside defining the response, we will have to ensure that the request is properly logged, which we will use the built in python library `logging` for. We create a file, `http.log`, which we will use for both our HTTP and HTTPS servers. The complete HTTP program can be seen in listing 5.6.

Listing 5.6: Simple python HTTP logging server

```
#!/usr/bin/env python

import SimpleHTTPServer
import SocketServer
import logging
logging.basicConfig(filename='http.log', level=logging.DEBUG)

PORT = 80

class GetHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_GET(self):
        logging.info(self.headers)
        self.send_response(200)
        self.end_headers()
        self.wfile.write('200 OK')

Handler = GetHandler
httpd = SocketServer.TCPServer(("", PORT), Handler)
httpd.serve_forever()
```

For HTTPS, we will use the previous HTTP server program as a base, and add the necessary functionality to wrap the connection in SSL. Python has a built in library for this too, and it is simply called `ssl`. By changing the port to 443, which is usually reserved for HTTPS, and wrapping the httpd socket in ssl we have the base functionality needed for HTTPS communication. To complete the HTTPS server we will need to create a SSL key and certificate to encrypt the communication. We will call these key.pem and cert.pem, and we can create them by running the command seen in listing 5.7.

Listing 5.7: Generate ssl key and certificate

```
openssl req -nodes -new -x509 -keyout key.pem -out cert.pem
```

Ensure that these keys are in the same directory as the HTTPS server program, or alternatively refer to them in the program by their absolute path in a location that python has read access to, and we have our HTTPS server ready as well, as seen in listing 5.8.

Listing 5.8: Simple python HTTPS logging server

```
#!/usr/bin/env python2

import SimpleHTTPServer
import SocketServer
import logging
import ssl

logging.basicConfig(filename='http.log',level=logging.DEBUG)
PORT = 443

class GetHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):

    def do_GET(self):
        logging.info(self.headers)
        self.send_response(200)
        self.end_headers()
        self.wfile.write('200 OK')


Handler = GetHandler
httpd = SocketServer.TCPServer(("", PORT), Handler)
httpd.socket = ssl.wrap_socket(httpd.socket,
        keyfile='key.pem',
        certfile='cert.pem', server_side=True)

httpd.serve_forever()
```

Now that we have both the server files, we need to run these in some stable manner. There are plenty of web server programs that the server functionality can be wrapped around, but by using the Unix command nohup and python itself, we can keep this running and active in the background without having to deal with any extra software. Nohup is a command that detaches the program it is given as an argument from the currently active session. This means that running the web servers with nohup will keep them independent of the active session, so they will keep running in the event that the active session is ended. To close them again you must find their process ID, and end it with regular unix tools, like kill. We also don't want any of the standard out output from the servers to clutter our active session, thus we will redirect any output to `/dev/null`. `/dev/null` can easiest be summarized as a black hole, as it is a special kind of file that erase anything that is sent to it. We follow up the pipe to /dev/null with `2>&1`, which tells the computer to redirect standard error to standard out, which is already sent to /dev/null, thus ensuring that all unnecessary output is sent to /dev/null and ignored. At the end of the commands we will append the & symbol, as this backgrounds the entire command. Thus we end up with the commands seen in listings 5.9 and 5.10 for running the HTTP and HTTPS services respectively.

Listing 5.9: Running the HTTP server

```
nohup python server.py > /dev/null 2>&1 &
```

Listing 5.10: Running the HTTPS server

```
nohup python secureserver.py > /dev/null 2>&1 &
```

Now both the HTTP and HTTPS server will log requests to the file http.log, so to verify callbacks we search through the http.log file for any request entries containing the subdomain used in the payload as the host header. An example of how a log entry will look after a http query can be seen in listing 5.11.

Listing 5.11: HTTP log example

```
INFO:root:Host: 9sr53tm13xikj2l0.vetle.io
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.97
Safari/537.36
Accept: image/webp,image/apng,image/*,*/*;q=0.8
dnt: 1
Referer: http://9sr53tm13xikj2l0.vetle.io/
Accept-Encoding: gzip, deflate
Accept-Language: nb,en-US;q=0.9,en;q=0.8,no;q=0.7,nn;q=0.6,da;q=0.5
```

5.4 LDAP

Most proper directory servers are installed with a fully fledged directory server software suite, including web servers and GUI's for interacting with the server. This will be unnecessary in our case, as we only need a barebones server, as we won't interact with the server much beyond querying its logs. For this reason I have chosen OpenLDAP, which is a open-source and lightweight implementation of the LDAP protocol. More specifically we will be using the `slapd` software, which is a stand-alone LDAP daemon using the OpenLDAP implementation of LDAP.

The actual package name for OpenLDAP in the Ubuntu package repositories is slapd, and we also need the package ldap-utils. To initiate the install of these packages we run the command `sudo apt install slapd ldap-utils`. We are then prompted with some configuration steps. The first step will prompt us to create a administrator password to use for later configuration. The rest of the steps provide default answers, which we can make use of as our installation doesn't have any specific demands that require further tweaking. After the install is completed, we want to configure the Directory Information Tree suffix to the domain we use. We run `sudo dpkg-reconfigure slapd`, and we give dealt answers to all steps except the one prompting us for the DNS domain name to use for constructing the base DN of the LDAP directory. In our case we supply vetle.io, which will construct the LDAP directory with a base DN of `dc=vetle, dc=io`.

For regular usage of OpenLDAP, this is the point where you would install some administration tool with a GUI to add, remove and edit records and properly manage the structure. We skip this step, as we are only gonna add some placeholder entry. Just to have some functional entry, we are gonna add a LDAP user `ldapuser` and the necessary hierarchical structure to contain it. To do this we create a `.ldif` file, which is the postfix of OpenLDAP entry files. The name does not matter, so we name it `data.ldif`, and its content for this system can be seen in listing 5.1. To add the contents of `data.ldif` to the OpenLDAP server we run the following command and authenticate with the OpenLDAP admin password:

```
ldapadd -x -W -D "cn=admin,dc=vetle,dc=io" -f ldapuser.ldif
```

Beyond this we do not really need to add any more entries to provide the functionality necessary for the system to run properly and log all queries. Since all queries are logged, we do not need to ensure that all queries return some entry, as LDAP will simply respond that no records were found to any queries that do not match any entries. If the system built in this thesis is being used to verify some specific vulnerability and a callback has been registered, this flexible configuration of OpenLDAP will allow us to simply add any custom entries if we want any callbacks to retrieve specific data. An example scenario where this is helpful is if we have some vulnerability that allows us to inject LDAP queries in a manner that makes the affected system attempt to retrieve data. In this scenario we can add a LDAP query containing some malicious data content as a payload, and craft the injected LDAP query to retrieve and potentially execute said payload.

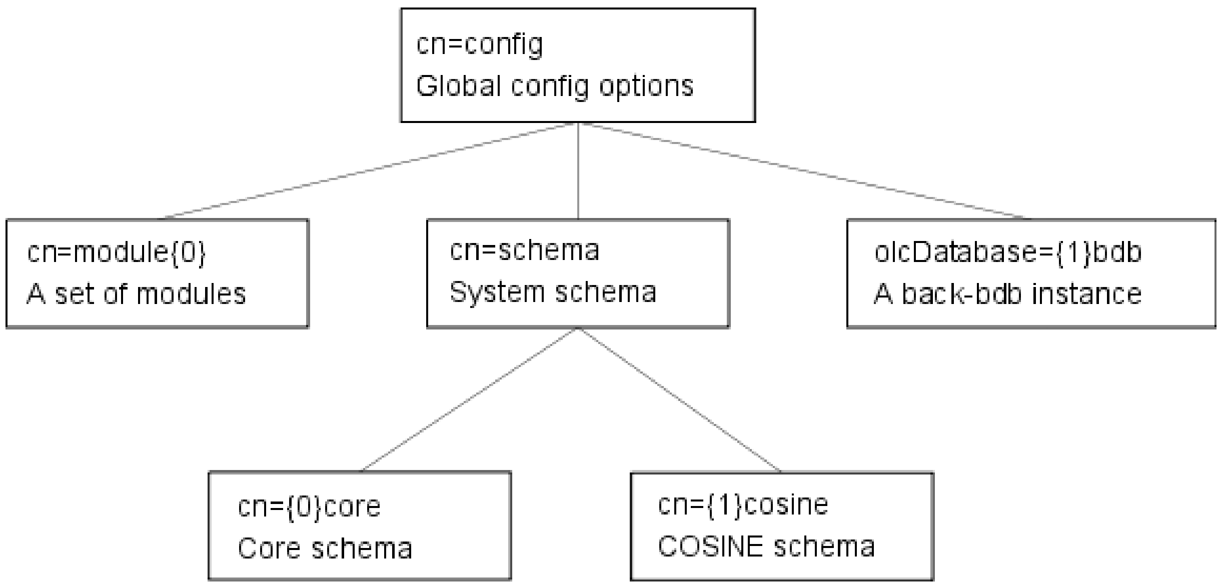
Listing 5.12: data.ldif

```
dn: ou=People,dc=vetle,dc=io
objectClass: organizationalUnit
ou: People

dn: ou=Group,dc=vetle,dc=io
objectClass: organizationalUnit
ou: Group

dn: uid=ldapuser,ou=People,dc=vetle,dc=io
objectClass: top
objectClass: account
objectClass: posixAccount
objectClass: shadowAccount
cn: ldapuser
uid: ldapuser
uidNumber: 9999
gidNumber: 100
homeDirectory: /home/ldapuser
loginShell: /bin/bash
gecos: Test LdapUser
userPassword: {crypt}x
shadowLastChange: 17058
shadowMin: 0
shadowMax: 99999
shadowWarning: 7
```

Figure 5.2: OpenLDAP sample config tree [15]



OpenLDAP has changed its configuration layout quite some times over the years, so a lot of the documentation found online are conflicting. This resulted in quite some effort spent on figuring out how to configure OpenLDAP to log the proper information, and how to log to a specific log file that we can access for searching. Earlier versions of OpenLDAP specified its configuration in a `slapd.conf` file, but since we are using the newest stable version there are a few more steps to changing the configuration. By reading the documentation we find that "The slapd configuration is stored as a special LDAP directory with a predefined schema and DIT. There are specific objectClasses used to carry global configuration options, schema definitions, backend and database definitions, and assorted other items." [15]. The documentation also provides a sample config tree, as seen in figure 5.2. To change the configurations we have to interact through LDAP with the cn=config LDIF. To achieve this we have to use the tools ldapsearch and ldapmodify. ldapsearch allows us to send some ldap query to the OpenLDAP instance to, for example, ask for information regarding the ldapuser user we created earlier. ldapmodify is used to modify any entries in the LDAP directory trees. First we want to figure out what kind of logging scheme is already in effect. To find this we run the query in listing 5.7.

Listing 5.13: ldapsearch query

```
sudo ldapsearch -Y external -H ldapi:/// -b cn=config
"(objectClass=olcGlobal)" olcLogLevel -LLL
```

From a default OpenLDAP installation this query should yield the response seen in listing 5.8. olcLogLevel determines what kind of events are logged, and with it set to none, OpenLDAP will not log anything.

Listing 5.14: olcLogLevel query response

```
dn: cn=config
olcLogLevel: none
```

To edit this, we need to create a LDIF file as we did when adding the ldapuser user to the base LDAP tree. We want the olcLogLevel set to stats, as this gives some basic information about the usage OpenLDAP sees, which includes queries and what elements where queried for. To achieve this we create the LDIF file seen in listing 5.9. The name does not matter, so let's call it loglevel.ldif.

Listing 5.15: loglevel.ldif

```
dn: cn=config
changeType: modify
replace: olcLogLevel
olcLogLevel: stats
```

We then apply this edit by running the command in listing 5.10.

Listing 5.16: ldapmodify command

```
sudo ldapmodify -Y external -H ldapi:/// -f loglevel.ldif
```

Now we have configured OpenLDAP to log basic usage statistics, but we have not yet specified where it should be logged to. Recent versions of OpenLDAP does not perform the actual logging themselves, but forwards the log information to rsyslog, which is a Unix syslog server. Rsyslog is highly flexible with logging rules, and to configure it to log to a specified location we have to create a slapd file in `/etc/rsyslog.d/`. This folder contains logging rules for services making use of rsyslog. Convention calls for calling the file some number, followed by a hyphen, then the name of the service .conf. So we create the file as `10-slapd.conf`, and add the content seen in listing 5.11.

Listing 5.17: 10-slapd.conf

```
$template slapdtmpl,"[%$DAY%-%$MONTH%-%$YEAR%
%timegenerated:12:19:date-rfc3339%] %app-name%
%syslogseverity-text% %msg%\n"
local4.*      /var/log/openldap/openldap.log;slapdtmpl
```

We must also ensure that the folder and file specified in 10-slapd.conf exists, and if not we must create it and ensure it has correct permissions. 10-slapd.conf makes use of rsyslogs presentation templates to create a generic and consistent logging format. After creating this file we restart the rsyslog server with `sudo service rsyslog restart`.

To check that the logging is working as intended, we can query OpenLDAP for some arbitrary entry in the vetle.io dc. An example query, and the following log output in `/var/log/openldap/openldap.log` can be seen in listing 5.12 and 5.13 respectively.

Listing 5.18: ldapsearch example query

```
ldapsearch -x cn=some_name -b dc=vetle,dc=io -H ldap://<server IP>
```

Listing 5.19: openldap.log output

```
[14-06-2020 18:25:37] slapd debug  conn=1004 op=0 BIND dn=""
method=128
[14-06-2020 18:25:37] slapd debug  conn=1004 op=0 RESULT tag=97
err=0 text=
[14-06-2020 18:25:37] slapd debug  conn=1004 fd=12 ACCEPT from
IP=<client IP>:7576 (IP=0.0.0.0:389)
[14-06-2020 18:25:37] slapd debug  conn=1004 op=1 SRCH
base="dc=vetle,dc=io" scope=2 deref=0 filter="(cn=some_name)"
[14-06-2020 18:25:37] slapd debug  conn=1004 op=1 SEARCH RESULT
tag=101 err=0 nentries=0 text=
[14-06-2020 18:25:37] slapd debug  conn=1004 op=2 UNBIND
[14-06-2020 18:25:37] slapd debug  conn=1004 fd=12 closed
```

In listing 5.13 we can see both the client IP and the cn they queried for are logged, and by confirming that the subdomain used in our injection payload appears in the logs, we can establish that a LDAP query was successfully executed.

5.5 Web interface

Since Flask is a web framework built on python, we necessarily need python installed on our system. This can be downloaded and installed from the official python website (python.org), with documentation. Additionally flask is a python package, and not a part of the default python library, thus we need to install a python package manager. Since Flask is in a public repository, we can make use of the official python package repository which is called PyPI (Python Package Index). Interacting with PyPI is done through the tool pip, which depending on the python version in use might come bundled with the python install. For python versions later than 2.7 it comes bundled, and earlier versions requires us to install it manually. Since the python version in use for this thesis project is 2.7, we have to manually install pip, which can easily be done by downloading and executing the official pip install script from `https://bootstrap.pypa.io/get-pip.py` . Preferably after applying best practice paranoia, and verifying that the code we run is the actual code intended to install pip.

When using python packages, you might run into the problem of developing applications that rely on different versions of dependencies. This poses a challenge, as maintaining multiple versions of packages for different applications is a non-trivial challenge to administrate. Luckily the solution is quite simple: virtual environments. By creating a virtual environment for our application to run inside, we get a complete copy of the python interpreter which runs completely separate from the system wide one. This means that any and all actions taken on packages within the virtual environment will be specific to said environment, and not affect the system wide python environment. This also applies the other way, as updating or changing the system wide python packages will not affect the ones installed within the virtual environment.

Considering the development build of this master thesis project is the only application running on the server, this step is technically optional, but if the system is intended to run alongside other python-based applications, virtual environments will save the system administrator a lot of maintenance work.

Installing and running a virtual environment will also depend on the version of python you run. If a python 3.5 version or later is in use, virtual environment support comes bundled and can be called as a module from python. For 3.4 and earlier, including our 2.7 version, we will make use of a third party tool called virtualenv. This is installed through the regular packet manager for the Linux distribution in use, in this case through `sudo apt install virtualenv` . We can then create the virtual environment in the folder we intend to build our application in, by running `virtualenv env` . This will create a file structure under the subfolder `./venv/`, which contains the actual files necessary for the virtual environment to work. At this point we have a virtual environment ready, but we are not yet using it. This means that any changes made to packages and python itself will still affect the system wide python interpreter. To activate the virtual environment and isolate the project from the system-wide python interpreter we run `source venv/bin/activate` in the folder of our project. Now the terminal session is modified so that the stored python interpreter is the one that's called when running the `python` command, ensuring that the correct interpreter and packages are in use. The terminal prompt will also be modified to include the name of the virtual environment, which in this case is venv. Running the `virtualenv` command with some other string as an argument would create a virtual environment with a different name, but otherwise has identical purpose.

With both python and a virtual environment installed and set up, we can now install the actual framework, flask. This is done by downloading it from PyPI, through running `pip install flask`. We also want to install the package `python-dotenv` which will allow us to save configuration information for the flask server in a file, instead of having to specify it on startup every time we launch flask.

To make use of flask and actually create a working web interface we will have to set up a basic directory structure. To ease the explanation I will visualize the directories and files necessary for basic functionality in listing 5.18.

Listing 5.20: flask directory structure

```
web-interface/
    venv/
    interface/
        __init__.py
        routes.py
        functions.py
        .env
    app.py
```

We have already gone through the venv functionality, so we will not repeat that aspect of the working directory. interface/ folder contains two important files. __init__.py functions as a initializer for the flask application, while routes.py contains what actions are to be taken dependent on which url is requested. functions.py contains the python functions necessary for interacting with the other modules of the system, to create subdomains,parse logs and retrieve log entries matching the searched subdomains. `.env` is a file containing the configuration for the flask server to use when running. The contents of app.py, __init__.py and .env can be seen in listings 5.20, 5.21 and 5.22 respectively, while the code for routes.py and functions.py can be found in the appendix.

Listing 5.21: app.py

```
from interface import interface
```

Listing 5.22: __init__.py

```
from flask import Flask

interface = Flask(__name__)

from interface import routes
```

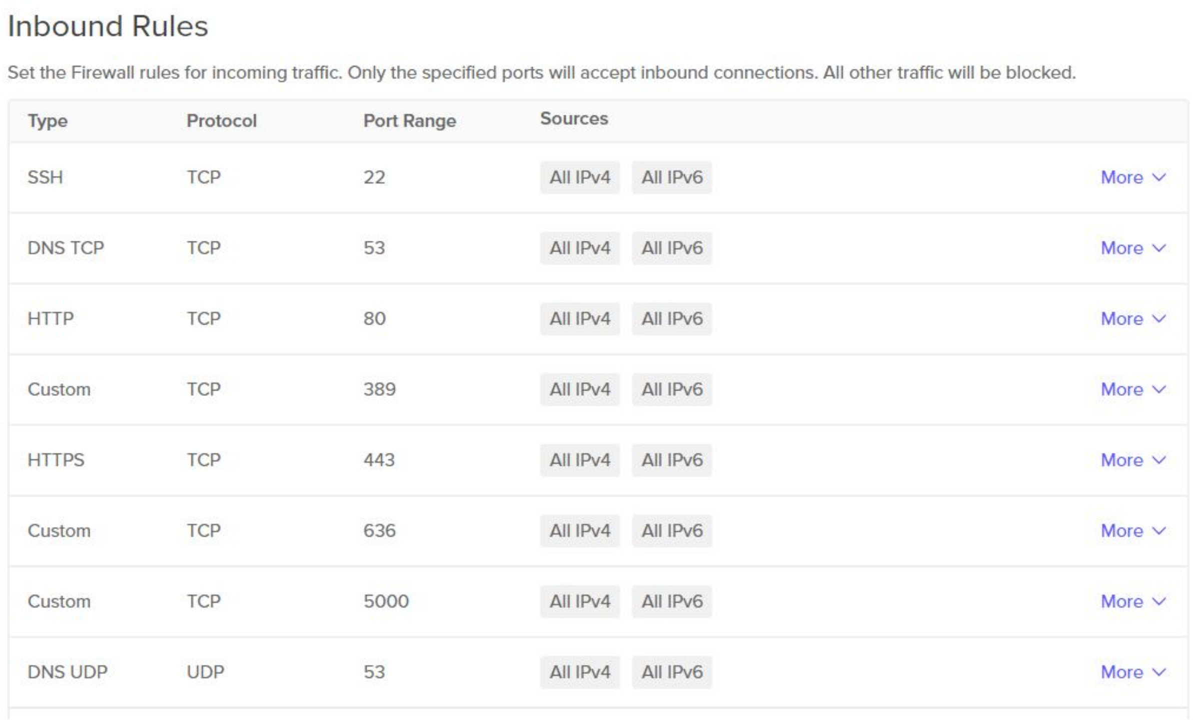
Listing 5.23: .env

```
FLASK_APP=interface
FLASK_ENV=development
FLASK_RUN_PORT=5000
FLASK_RUN_HOST=0.0.0.0
```

With the simple web interface functionality specified, we have a web interface that allows us to create a subdomain as a random string value, and we can search through logs for the implemented protocols. To actually run the website, in the web-interface directory, we run the command `flask run`. This can also be replaced by flask deployment server alternatives, but for testing and development purposes we will use the built in server. Since both port 80 and 443 are used for the HTTP and HTTPS logging services, port 5000 was chosen for the web interface since it is the default development port for flask. Which port is used does not matter as long as it is a high-level port not reserved by some other protocol.

5.6 Firewall

Both for development and secure operation of the tool, it should be firewalled so that only whitelisted clients can access it. During development I have chosen to do implement a firewall through DigitalOcean's built-in droplet configuration. For inbound firewall rules I allow ports 22 (ssh), 53 (DNS), 80 (HTTP), 443 (HTTPS), 389 (LDAP), 636 (LDAP over SSL) and 5000 (development web interface) over TCP, and 53 (DNS) over UDP.

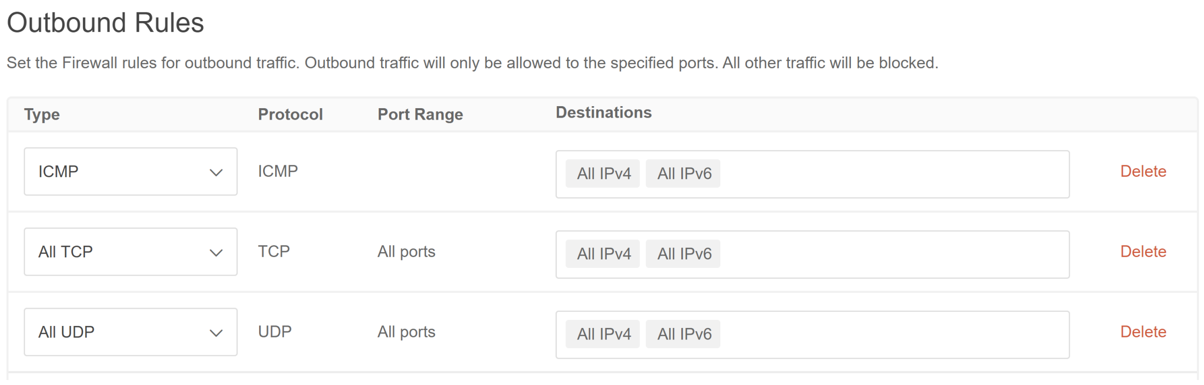


Inbound Rules

Set the Firewall rules for incoming traffic. Only the specified ports will accept inbound connections. All other traffic will be blocked.

Type	Protocol	Port Range	Sources	
SSH	TCP	22	All IPv4 All IPv6	More
DNS TCP	TCP	53	All IPv4 All IPv6	More
HTTP	TCP	80	All IPv4 All IPv6	More
Custom	TCP	389	All IPv4 All IPv6	More
HTTPS	TCP	443	All IPv4 All IPv6	More
Custom	TCP	636	All IPv4 All IPv6	More
Custom	TCP	5000	All IPv4 All IPv6	More
DNS UDP	UDP	53	All IPv4 All IPv6	More

For outbound I allow all ICMP, TCP and UDP. To comply with security best practices this should be restricted to only the outbound connections necessary for the tool to function, but I keep it open during development to not block any misconfigured server responses or similar errors.



Outbound Rules

Set the Firewall rules for outbound traffic. Outbound traffic will only be allowed to the specified ports. All other traffic will be blocked.

Type	Protocol	Port Range	Destinations	
ICMP	ICMP		All IPv4 All IPv6	Delete
All TCP	TCP	All ports	All IPv4 All IPv6	Delete
All UDP	UDP	All ports	All IPv4 All IPv6	Delete

Choosing to use DigitalOceans built-in firewall restricts the security measures to droplets deployed through DigitalOcean. Regardless, the choice of firewall is in this case mostly a matter of preference. Any host-based firewall, or additional network perimeter firewall, could be used as long as it provides basic firewall functionality of specifying open ports, and denying all others. As it is certain that any network connected system will be the target of scanning and probing from all kinds of actors on the net, establishing a firewall in this manner will help both alleviate any performance load increase, and also reduce the digital footprint and attack surface of the tool itself. A firewall should not be depended on exclusively, but using it as a tool in addition to regular best practice secure software development practices and operations techniques, it greatly increases the overall security of the system.

Chapter 6

Results

In the beginning of the project, we set out to design and build a modular callback registering system for DNS, HTTP(S) and LDAP. The design of the system turned into building a centerpiece consisting of a Flask web interface, creating stand-alone installations of the selected implementations of the protocols selected, and adding log parsing support for those services to the web interface centerpiece. The Bind9 DNS service also became an essential part together with the web interface, as the DNS service is responsible for actually creating the subdomains used to separate which callbacks are to be parsed and presented. Both the HTTP(S) and LDAP modules could be removed, and the system would still work perfectly fine for DNS callback registering, but if the DNS module is removed then most of the functionality would break. This dependency happens because in it's current implementation the DNS server acts as the authoritative nameserver for the selected domain, and if the Bind9 instance is removed then DNS resolution would not work, and callbacks would be unable to resolve any domains. Failing to resolve the domain means that the vulnerable server would never be able to reach our system, regardless if the attack was successful or not.

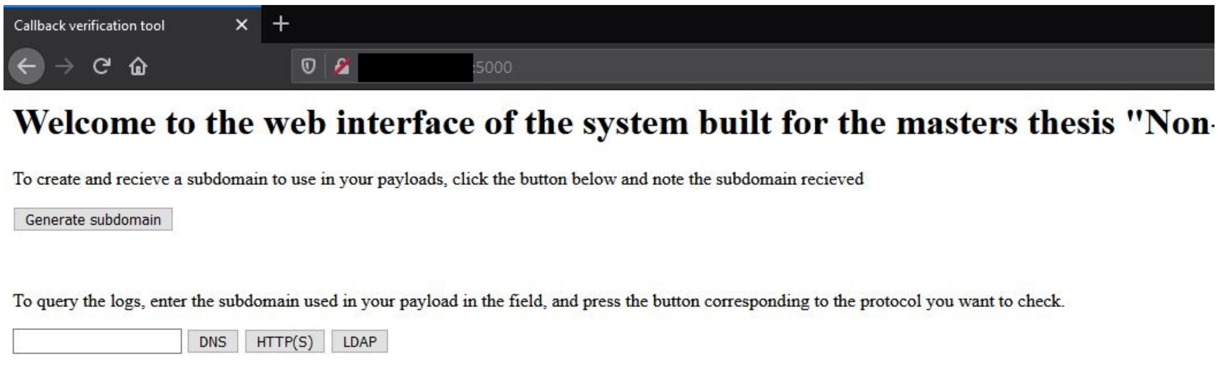
Technically this doesn't stop us from merely crafting payloads that refer directly to the IP of the server, but by doing this we lose the opportunity to see which queries in the logs were actually the callback. As the DNS module creates a random subdomain which is used in the payloads, the web interface uses this subdomain when parsing the relevant logs to find the log entries related to that specific payload. Without this identifier, any callbacks would be tough to isolate within the logs. Any system that is exposed to the internet will receive probing requests and scans all the time, thus our callbacks would very likely drown in the "noise" from the internet at large.

During the actual development phase of the system, making sure that the Bind9 server operated in a stable and predictable manner turned into a much greater challenge than initially expected. At the beginning of the project, the complexity of doing dynamic DNS updates was underestimated, and while there exists documentation and tutorials for how to implement and manage a Bind9 instance, doing dynamic subdomain creation in the manner this system demanded was not covered in much detail. This resulted in much time being spent on experimenting with the configuration of Bind9, and how the python functions called from the web interface did its DNS update queries.

While experimenting with how to do dynamic DNS queries, lots of testing was done by manually editing the zone file, which resulted in many frustrated hours dealing with out of sync journal errors, as mentioned in chapter 5.2. Bind9 has functionality allowing for ”freezing” and ”thawing” zones when doing manual updates, but how to translate that functionality into automated subdomain creation was not straightforward. The constant instability of that approach resulted in a standard DNS update query followed by a restart to be chosen as the simpler and more stable solution.

Some screenshots of the system in action can be seen in the figures 6.1-5.

Figure 6.1: Index page of web interface

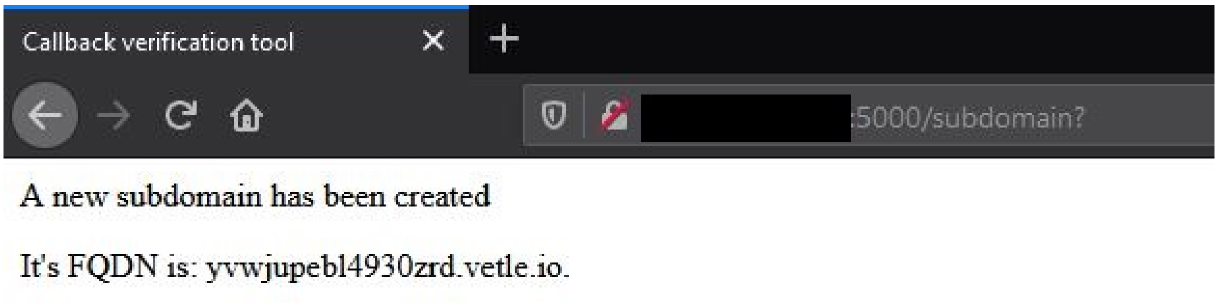


Welcome to the web interface of the system built for the masters thesis "Non

To create and recieve a subdomain to use in your payloads, click the button below and note the subdomain recieved

To query the logs, enter the subdomain used in your payload in the field, and press the button corresponding to the protocol you want to check.

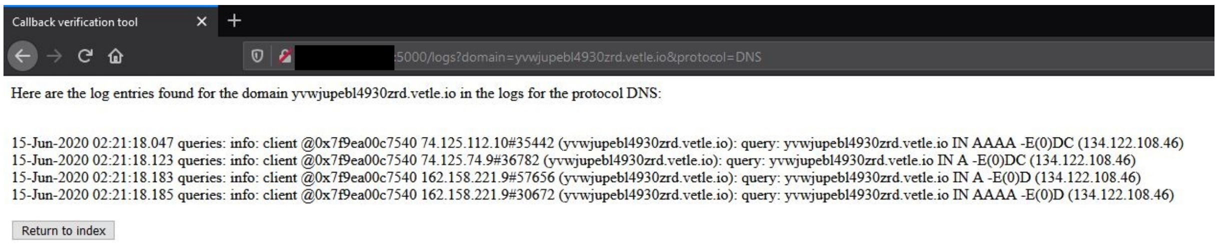
Figure 6.2: Web interface page generating new subdomain



A new subdomain has been created

It's FQDN is: yvwjupebl4930zrd.vetle.io.

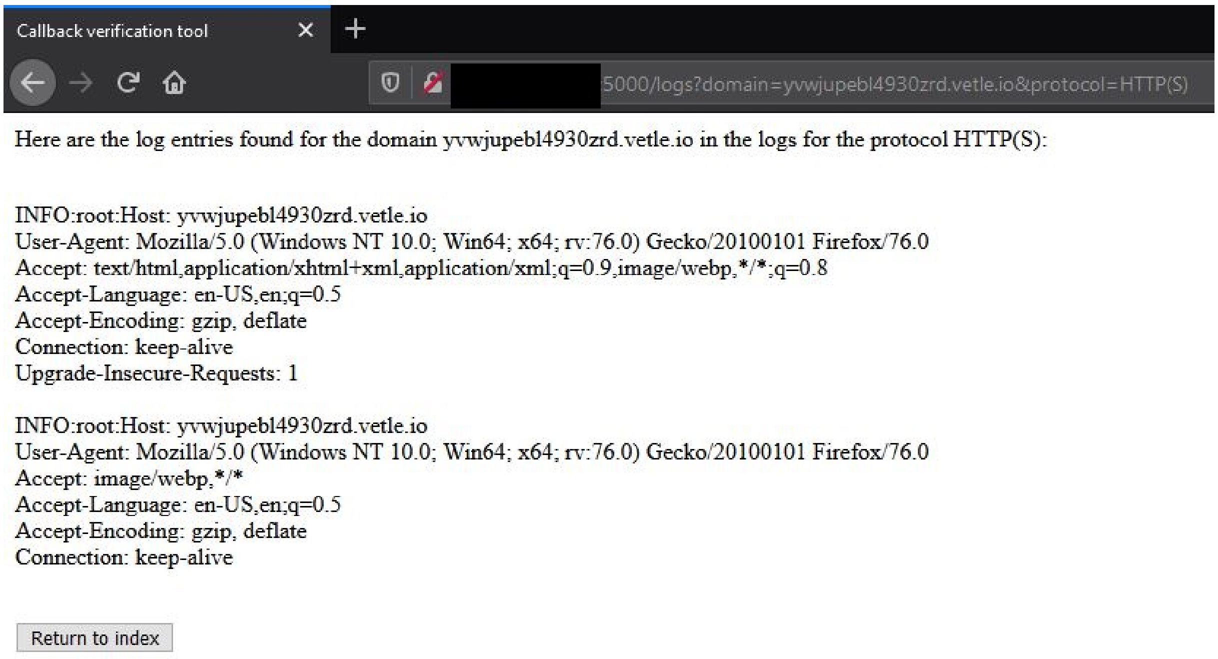
Figure 6.3: Log page returning matching DNS logs


Callback verification tool
[redacted]:5000/logs?domain=yvwjupebl4930zrd.vetle.io&protocol=DNS

Here are the log entries found for the domain yvwjupebl4930zrd.vetle.io in the logs for the protocol DNS:

```
15-Jun-2020 02:21:18.047 queries: info: client @0x7f9ea00c7540 74.125.112.10#35442 (yvwjupebl4930zrd.vetle.io): query: yvwjupebl4930zrd.vetle.io IN AAAA -E(0)DC (134.122.108.46)
15-Jun-2020 02:21:18.123 queries: info: client @0x7f9ea00c7540 74.125.74.9#36782 (yvwjupebl4930zrd.vetle.io): query: yvwjupebl4930zrd.vetle.io IN A -E(0)DC (134.122.108.46)
15-Jun-2020 02:21:18.183 queries: info: client @0x7f9ea00c7540 162.158.221.9#57656 (yvwjupebl4930zrd.vetle.io): query: yvwjupebl4930zrd.vetle.io IN A -E(0)D (134.122.108.46)
15-Jun-2020 02:21:18.185 queries: info: client @0x7f9ea00c7540 162.158.221.9#30672 (yvwjupebl4930zrd.vetle.io): query: yvwjupebl4930zrd.vetle.io IN AAAA -E(0)D (134.122.108.46)
```

Figure 6.4: Log page returning matching HTTP(S) logs

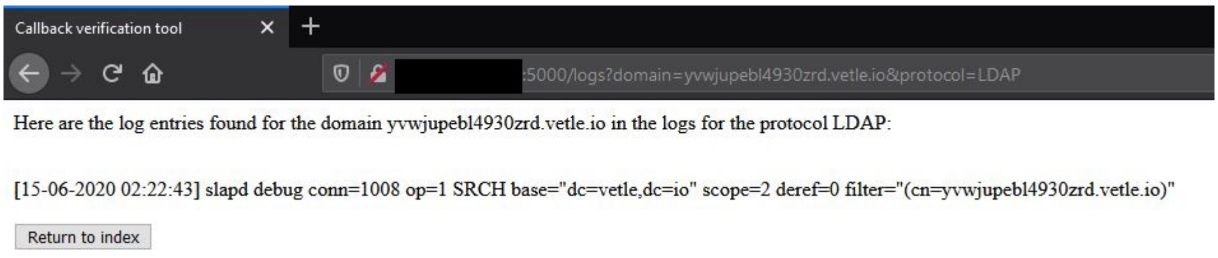

Callback verification tool
[redacted]5000/logs?domain=yvwjupebl4930zrd.vetle.io&protocol=HTTP(S)
Return to index

Here are the log entries found for the domain yvwjupebl4930zrd.vetle.io in the logs for the protocol HTTP(S):

```
INFO:root:Host: yvwjupebl4930zrd.vetle.io
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:76.0) Gecko/20100101 Firefox/76.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1

INFO:root:Host: yvwjupebl4930zrd.vetle.io
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:76.0) Gecko/20100101 Firefox/76.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Figure 6.5: Log page returning matching LDAP logs


Callback verification tool
[redacted]5000/logs?domain=yvwjupebl4930zrd.vetle.io&protocol=LDAP
Return to index

Here are the log entries found for the domain yvwjupebl4930zrd.vetle.io in the logs for the protocol LDAP:

```
[15-06-2020 02:22:43] slapd debug conn=1008 op=1 SRCH base="dc=vetle,dc=io" scope=2 deref=0 filter="(cn=yvwjupebl4930zrd.vetle.io)"
```

Chapter 7

Conclusion

The primary goal for this project starting out was to build a system capable of performing the actions outlined in the Introduction chapter. This goal was met, as the system is capable of creating arbitrary subdomains and logging any callbacks towards it from DNS, HTTP(S) and LDAP. The drawback is that the current system has no simple method of installation and requires quite extensive configuration of all the programs it depends on. Further efforts on improving this system would have to be made towards creating either some install script that manages to build and configure all required modules automatically, or a software package for user repositories would have to be built.

Verifying blind injection vulnerabilities through "out of band" network traffic proves to be a quite robust method of non-destructive verification, as all network accessible services usually has a functional network stack that will allow it to make network calls without major risk for crashing or other erratic behavior. Of course this is highly dependant on the system in question, but "out of band" verification has a higher chance of not affecting the system than many of the alternatives. Examples are time-based verification, which can be especially intrusive to systems that perform actions in a synchronous manner, so a payload that holds up the current action might severely affect the later actions. "Out of band" verification also has the added benefit of not requiring changing anything on the service being tested to work, as it will merely perform network calls, as opposed to, for example, leaving some artefact on the server to prove a successful PoC.

Secondarily, a goal was outlined for learning more about the protocols and technologies involved in the system. Starting out, my familiarity with DNS was limited to a theoretical understanding of the protocol, and from managing a domain indirectly through ISP's. Actually installing, configuring and managing a nameserver directly demanded thorough research and improved my knowledge within both the outlined specifications of the protocol, and with the technical challenges present from running a nameserver. This led me to a greater understanding of why DNS acts as it does, and why there is no simple way to "merely improve" DNS as a backbone for the internet, even in the face of the security challenges present.

HTTP(S) was a protocol quite well known to me already, and I have had quite some experience with running webservers for a multitude of purposes. Thus hosting a web server and recording the necesarry logging was one of the more straightforward parts of the research and effort going into this thesis.

While I have had some experience with Active Directory, the specifics of LDAP was not a well known subject to me. Setting up a stand-alone simple OpenLDAP server, configuring it to reply and act according to the protocols specifications, and ensuring that it logged the correct queries and events gave me a much deeper understanding of how directory servers are used in modern organization environments. Such knowledge will be highly beneficial in future security audits and testing of on-premise computer systems, as organizations often are highly reliant on their chosen system of directory services and authentication.

The current build of the system can be considered a promising prototype for a highly modular and flexible callback registering system. There is still some work left regarding stability and ease-of-installation before this can be considered ready to be released as a tool. This future work would also include a more user-friendly web interface, or some other alternative way of interacting with the system.

There was significant technical knowledge and expertise gained during the process of building this system, and I am quite certain that this will help me in my future endeavors as a computer security professional.

References

[1] Merriam-Webster. *Hacker.* URL: https://www.merriam-webster.com/dictionary/hacker (visited on 06/08/2020).

[2] Inc. Cloudflare. *What is Penetration Testing? What is Pen Testing?* URL: https://www.cloudflare.com/learning/security/glossary/what-is-penetration-testing/ (visited on 06/08/2020).

[3] Inc. Comodo Group. *Computer Vulnerability: Definition.* URL: https://enterprise.comodo.com/blog/computer-vulnerability-definition/ (visited on 06/08/2020).

[4] Rapid7. *Vulnerabilities, Exploits, and Threats.* URL: https://www.rapid7.com/fundamentals/vulnerabilities-exploits-threats/ (visited on 06/08/2020).

[5] Inc. Cloudflare. *What is A Malicious Payload?* URL: https://www.cloudflare.com/learning/security/glossary/malicious-payload/ (visited on 06/08/2020).

[6] The Linux Information Project. *Daemon Definition.* URL: http://www.linfo.org/daemon.html (visited on 06/08/2020).

[7] w3schools.com. *SQL Injection.* URL: https://www.w3schools.com/sql/sql_injection.asp (visited on 06/08/2020).

[8] Jeff Forristal. *About Me.* URL: https://www.forristal.com/about.html (visited on 05/29/2020).

[9] rain.forest.puppy / [WT]. "NT Web Technology Vulnerabilities." In: *Phrack Magazine* 8.54 (1998). URL: http://phrack.org/issues/54/8.html.

[10] Stuart McDonald. "SQL Injection: Modes of Attack, Defence, and Why It Matters." In: *SANS Institute Information Security Reading Room* (2002), p. 22. URL: https://www.sans.org/reading-room/whitepapers/securecode/sql-injection-modes-attack-defence-matters-23.

[11] Open Web Application Security Project. *Testing for SQL Injection.* URL: https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection.html (visited on 06/11/2020).

[12] Open Web Application Security Project. *SQL Injection.* 2019. URL: https://owasp.org/www-community/attacks/SQL_Injection (visited on 05/29/2020).

[13] Open Web Application Security Project. *Blind SQL Injection*. 2019. URL: `https://owasp.org/www-community/attacks/Blind_SQL_Injection` (visited on 05/29/2020).

[14] Neil Wilson. *Learn About LDAP*. URL: `https://ldap.com/learn-about-ldap/` (visited on 06/04/2020).

[15] OpenLDAP Foundation. *Configuring slapd*. URL: `https://www.openldap.org/doc/admin24/slapdconf2.html` (visited on 06/14/2020).

Appendix

Listing 7.1: /etc/bind/named.conf.options

```
options {
        directory "/var/cache/bind/";

        recursion no;
        allow-transfer { none; };
        allow-query { any; };
        querylog yes;

        forwarders {
                8.8.8.8;
                8.8.4.4;
        };

        dnssec-validation auto;

        auth-nxdomain no;
        listen-on-v6 { any; };
};

logging {
    channel query_log {
        file "/var/log/named/query.log" versions 3 size 5m;
        print-category yes;
        print-severity yes;
        print-time yes;
        severity dynamic;
        };
    channel update_debug {
        file "/var/log/named/update_debug.log" versions 3 size 5m;
        severity debug ;
        print-category yes;
        print-severity yes;
        print-time yes;
        };
    channel security_info {
        file "/var/log/named/security_info.log" versions 3 size 5m;
        severity info;
        print-category yes;
        print-severity yes;
        print-time yes;
        };
    channel bind_log {
        file "/var/log/named/bind.log" versions 3 size 5m;
        severity info;
        print-category yes;
        print-severity yes;
        print-time yes;
        };
    category queries {
        query_log;
        };
    category security {
        security_info;
        };
    category update-security {
        update_debug;
        };
    category update {
        update_debug;
        };
    category lame-servers {
        null;
        };
    category default {
        bind_log;
        };
};
```

Listing 7.2: routes.py

```
from interface import interface
from flask import request
import functions

@interface.route('/')
@interface.route('/index')
def index():
   return '''
<html>
    <head>
        <title>Callback verification tool</title>
    </head>
    <body>
        <h1>Welcome to the web interface of the system built for
        the masters thesis "Non-desctructive verification of blind
        injection attacks"</h1>
        <p>To create and recieve a subdomain to use in your
        payloads, click the button below and note the subdomain
        recieved</p>
        <form method="get" action="/subdomain">
            <button type="submit">Generate subdomain</button>
        </form>
        <br>
        <p>To query the logs, enter the subdomain used in your
        payload in the field, and press the button corresponding to
        the protocol you want to check. </p>
        <form method="get" action="/logs">
            <input type="text" name="domain"/>
            <input type="submit" name="protocol" value="DNS"></input>
            <input type="submit" name="protocol" value="HTTP(S)"></input>
            <input type="submit" name="protocol" value="LDAP"></input>
        </form>
    </body>
</html>'''

@interface.route('/subdomain')
def subdomain():
    subdomain = functions.createSubdomain()
    return '''
<html>
    <head>
```

```
        <title>Callback verification tool</title>
    </head>
    <body>
        <p>A new subdomain has been created</p>
        <p>It's FQDN is: ''' + subdomain + '''</p>
        <form method="get" action="/">
            <button type="submit">Return to index</button>
        </form>
    </body>
</html>'''

@interface.route('/logs')
def logs():
    domain = request.args.get('domain', default ='*', type = str)
    protocol = request.args.get('protocol', default = '*', type = str)
    if protocol == 'DNS':
        result = functions.dnsQueries(domain)
    elif protocol == 'HTTP(S)':
        result = functions.httpQueries(domain)
    elif protocol == 'LDAP':
        result = functions.ldapQueries(domain)
    else:
        return '''
<html>
    <head>
        <title>Callback verification tool</title>
    </head>
    <body>
        <p>No applicable protocol was selected</p>
        <form method="get" action="/">
            <button type="submit">Return to index</button>
        </form>
    </body>
</html>'''
    return '''
<html>
<head>
        <title>Callback verification tool</title>
    </head>
    <body>
        <p>Here are the log entries found for the domain ''' +
        domain + ''' in the logs for the protocol ''' + protocol +
        ''':</p>
        <br>''' + result + '''<br>
```

```
            <form method="get" action="/">
                <button type="submit">Return to index</button>
            </form>
        </body>
</html>'''
```

Listing 7.3: functions.py

```python
import random
import string
import os
import sys
import time
import dns.update
import dns.query
import dns.tsigkeyring
import re

dnslogs = '/var/log/named/query.log'
httplogs = '/root/thesis-project/http/http.log'
ldaplogs = '/var/log/openldap/openldap.log'

keyring = dns.tsigkeyring.from_text({
    'DDNS_UPDATE.' : 'RgRvLyeJDr54Feq5bDyXzw=='
})

def randomString(stringLength=16):
    characters = string.ascii_lowercase + string.digits
    return ''.join((random.choice(characters) for i in
        range(stringLength)))

def createSubdomain():
    subdomain = randomString()
    FQDN = subdomain + '.vetle.io.'
    zone = "vetle.io"
    update = dns.update.Update(zone, keyring=keyring)
    update.add(FQDN, 8600, 'A', '134.122.108.46')
    response = dns.query.tcp(update, '127.0.0.1', timeout=10)
    time.sleep(0.5)
    os.system("sudo /etc/init.d/bind9 restart")
    return FQDN

def dnsQueries(FQDN):
    results = ""
    for line in open(dnslogs, 'r'):
        if re.search(FQDN, line):
            results = results + line + "<br>"
    return results
```

```
def httpQueries(FQDN):
    results = ""
    toggle = False
    for line in open(httplogs, 'r'):
        if re.search(FQDN, line):
            toggle = True
        if toggle == True:
            results = results + line + "<br>"
        if line == "":
            toggle == False
            results = results + "<br>"

    return results

def ldapQueries(FQDN):
    results = ""
    for line in open(ldaplogs, 'r'):
        if re.search(FQDN, line):
            results = results + line + "<br>"
    return results
```