TU WIEN Informatics

Automatic Creation of Low-Interaction Honeypots for Stateless Network Protocols

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Felix Leopold Kehrer
Matrikelnummer 01526278

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Dipl.-Ing. Christian Kudera, BSc
Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc
Univ.Lektor Dipl.-Ing. Michael Pucher, BSc

Wien, 19. Oktober 2022

Felix Leopold Kehrer	Edgar Weippl

TU WIEN Informatics

Automatic Creation of Low-Interaction Honeypots for Stateless Network Protocols

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Felix Leopold Kehrer
Registration Number 01526278

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Christian Kudera, BSc
Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc
Univ.Lektor Dipl.-Ing. Michael Pucher, BSc

Vienna, 19th October, 2022

Felix Leopold Kehrer | Edgar Weippl

Erklärung zur Verfassung der Arbeit

Felix Leopold Kehrer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Oktober 2022

Felix Leopold Kehrer

Danksagung

Ich danke meinen Betreuern Christian Kudera, Georg Merzdovnik und Micheal Pucher für ihre Anregungen und ihr Feedback. Außerdem danke ich Jakob Bleier für seine Schreibratschläge und sein Feedback.

Acknowledgements

I thank my advisors Christian Kudera, Georg Merzdovnik, and Michael Pucher for their input and feedback. Additional thanks go to Jakob Bleier for his writing advice and feedback.

Kurzfassung

Low-Interaction Honeypots stellen immer einen Kompromiss zwischen guter Interaktivität und niedrigen Kosten dar. Es gibt bereits Programme die Low-Interaction Honeypots automatisch erstellen, aber die so erstellten Honeypots sind weniger interaktiv als sie sein müssten um vielseitig nützlich zu sein. Deshalb stellen wir einen Ansatz vor, mit dessen Hilfe interaktivere Honeypots erstellt werden können, ohne Abstriche bei der Nutzerfreundlichkeit in Kauf nehmen zu müssen. Dieser Ansatz agiert auf einer niedrigeren Ebene als bestehende, und ist gleichzeitig flexibel genug um auch für andere Protokolle außer HTTP(S) nützlich zu sein. Wir implementieren außerdem ein Programm namens SyrupPot, welches in der Lage ist automatisch Kopien von bestehenden HTTP(S) zu erstellen, die dann als Honeypots fungieren. Diese Honeypots können mehr Situationen bewältigen als bestehende Ansätze und lassen sich leicht an neue Anforderungen anpassen. Um ihre Nützlichkeit zu demonstrieren, erstellen wir Honeypots von IoT Geräten und untersuchen deren Beschaffenheit.

Abstract

Low-interaction honeypots are always a compromise between having good interactivity and being cheap to create and run. Tools exist to automatically create low-interaction honeypots, but they achieve their great usability at the cost of interactivity, limiting their usefulness. We introduce a way of achieving better interactivity while maintaining the same level of usability, which operates at a lower level and is flexible enough to be used with other protocols besides HTTP(S). We implement a tool called SyrupPot to create copies of existing HTTP(S) services automatically, which then server as honeypots. These honeypots can handle more situations than existing approaches and are easily adapted to new requirements. To demonstrate their usefulness we create honeypots from IoT devices and evaluate their quality.

Contents

CHAPTER 1

Introduction

Web services are everywhere. They are in charge of countless things, from enormous tasks like global finance to small conveniences like telling you whether it's warm or cold outside. Recently, there is a trend to make even the smallest sensor a little connected computer: The Internet of Things (IoT). With more and more devices being created and connected, their usefulness increases, but so does the risk. Large-scale malware campaigns are becoming increasingly common, while the devices they are targeting contain more and more sensitive information about their users. At the time of writing, a new malware called *Shikitega* was only just discovered, which targets Linux-based IoT devices[1]. As such, countermeasures are more important than ever.

In the context of a possible attack, a lot can be done to prepare. Whether that means making the device harder to contact for unauthorized actors, hardening the hard- and software against exploitable vulnerabilities, installing updates, setting better passwords, identifying and inhibiting attacks, feeding the attackers false information, or any number of other ways of making the attacker's job harder.

On the one hand, honeypots are a great tool for achieving some of these goals. They distract attackers from real targets, thereby wasting their time and resources, they allow for easy detection of attacks as they are happening, and can deliver high-quality information about such an attack. On the other hand, creating and running a convincing honeypot often involves a substantial amount of work and resources.[2]

To combat both of these drawbacks, tools exist to automatically create low-interaction honeypots, needing little manual intervention to create a honeypot, and few resources to run the same. Unfortunately, these tools have stark limitations, making the honeypots rather trivial to identify and thereby limiting their usefulness. In this paper, we present an approach for automatically creating low-interaction honeypots from existing devices or services that can overcome some of these limitations, and so lead to better low-interaction honeypots in the future.

But first, an overview:
Section 2 explains or defines some vocabulary relevant to the rest of the paper.
Section 3 talks about which related tools already exist and the problems that plague them.
Section 4 describes our approach, and the reasoning behind it.
Section 5 recounts the development of our tool, SyrupPot, the roadblocks we encountered, how we dealt with them, and the architecture we eventually settled on.
Section 6 goes over our evaluation setup and the results of the evaluation. And finally, section 7 summarises the content of the paper.

We understand our contributions as the following:

- We describe the existing approaches and tools for generating low-interaction honeypots automatically and identify problems holding back their usefulness.
- We introduce a more general approach to low-interaction honeypot generation, which includes collecting more data and using that data to make more convincing honeypots.
- We describe the tool we built to demonstrate the usefulness of our approach. It automatically creates copies of HTTP services, which can easily be deployed on any host system supporting Docker.
- We talk about the limitations of our tool, what can be done about those limitations, and what future improvements to the tool might look like.

CHAPTER 2

Background

2.1 Honeypots

A **honeypot** is a system that appears to be a normal system serving some normal purpose, but which is actually engineered to not be used at all during normal proceedings. As such, any attempt to interact with it is immediately suspicious. This concept applies rather broadly to many forms of systematic misinformation, including things like honeywords — fake passwords that, when used, alert the victims of a data leak that such a leak has happened. For our purposes, a honeypot means a network service that attempts to pass itself off as another network device or service, in order to distract attackers from valuable targets, identify ongoing attacks and collect as much information as possible on said attacks.

As such, the usefulness of a honeypot is directly linked to its convincingness, because the moment an attacker figures out that a system is a honeypot, they might stop interacting with it, try to feed it false information, or even attempt to break it, in order to hide their tracks.

There are two basic ways of identifying a honeypot. The first is by interacting with it. Vetterl and Clayton identify honeypots by specifically crafting probes to find out which specific honeypot is used[3], but for an attacker, any probe that can make the honeypot behave measurably different from a real system might do. The second way is to use context information. Sophisticated metrics like Shodan's Honeyscore[4] use a multitude of data points like the history of an IP address and the network topology in addition to probing to judge how likely an internet-connected device is to be a honeypot.[5]

A system that behaves (almost) indistinguishably from a normal system is called a **high-interaction honeypot**, meaning that the honeypot can respond convincingly to (almost) any request it receives. High-interaction honeypots usually range between normal production systems with fake data and emulations of such systems. But, such

an arrangement comes at a considerable cost in terms of resources, comparable to the cost of a normal production system, because fundamentally, they are production systems, just not used as such.

That's where **low-interaction honeypots** come in. The idea is to trade some level of interactivity for reduced resource and engineering requirements, by abstracting away parts of the actual implementation of a production system and only focusing on the parts that matter for an attack scenario. Because anything that an attacker cannot interact with is probably not worth emulating, these are usually simple web services, sharing the outwards appearance of a real service but offering reduced functionality, like not maintaining state between interactions, always refusing authentication attempts, or not offering any functionality that would require the presence of an operating system. This also makes them less of a risk because an attacker cannot overtake and use these honeypots like a real system because they work differently and lack many of the capabilities of a real system.

While low-interaction honeypots are much cheaper to create and operate than their high-interaction counterparts, they have two obvious downsides: Firstly, their limitations make them easier to identify because their behaviour differs more from a normal system. And secondly, should an attacker be fooled, they cannot give as much valuable information as a high-interaction honeypot. For example, if an attacker never manages to log in to a service, they cannot access the data they are interested in, and as such the people or systems monitoring the honeypot never find out what the attacker was after. These problems can be mitigated by making low-interaction honeypots better and more complex, but not fully overcome. Such an improved low-interaction honeypot is sometimes called a **medium-interaction honeypot**[2].

2.2 HTTP

HTTP, or HyperText Transfer Protocol, is one of the fundamental technologies of today's internet. As the name implies, its purpose is to transfer hypertext, meaning text documents that can link to other resources. These documents are displayed to users by a program called a *browser*. Over time HTTP has come to be used for more general web transport of data.

HTTP works by sending messages between two parties. The client sends a **request** message to the server, and the server answers with a **response** message. Multiple requests and responses can be sent over the same connection, but don't have to. Because HTTP does not depend on the state of the connection, it is called a **stateless** protocol. Any context information required is included in the messages. A collection of (related) HTTP request/response pairs is sometimes called an HTTP flow[6].

A request message consists of a **request line**, some **headers**, and optionally a message **body**. The request line specifies a resource and a method to apply to this resource. The headers specify additional information like which data encodings the client can

understand, expressed as key-value pairs. The body can contain arbitrary binary data. A response message works similarly, but instead of a request line, it starts with a *status line*, indicating whether the request could be fulfilled. It also uses different headers and a body is present in most cases.

The act of extracting information from a web page is referred to as scraping. Because a web page is, among other things, comprised of hypertext, this means it's possible to scrape the hyperlinks, follow them, and then scrape those pages as well. A program doing this would be called a **crawler** or a **spider** — because it "crawls the web".

CHAPTER 3

State of the art

3.1 Existing tools and approaches

Vetterl and Clayton[3] consider **Conpot**[7], **Dionaea**[8] and **Glastopf**[9] to be the state-of-the-art HTTP honeypots, and between those three recommend to use Glastopf. All three of them require the source code of a web page to function, so they cannot impersonate a web device out of the box. Instead one needs to extract the actual web page files from the device with another tool or manually. Vetterl and Clayton were also able to identify instances of all three honeypots in the wild based on quirks in their server implementations.

SNARE[10], together with its companion service **TANNER**[11] is the successor to Glastopf, providing all of Glastopf's original features, but also more: It includes a crawler and so can create a copy of a website automatically. It also has a new server implementation based on aiohttp[12] which can use additional information saved by the crawler to better impersonate the original service. TANNER on the other hand handles the vulnerability type emulation that was also one of Glastopf's features. It essentially tells SNARE how to respond to requests.

CHAMELEON[13] does many similar things but has a different focus. The idea behind CHAMELEON is to identify dynamic content on a page and generate templates that can replace these dynamic parts with other, fitting data. For instance, timestamps are recognized and updated accordingly. Even data that is sent in the request and then appears in the response is identified. Unfortunately, CHAMELEON is not publicly available.

Another tool worth mentioning here is **Honeyd**[14]. Instead of attempting to impersonate a device on the service level, its focus is on replicating characteristics of an OS, in particular the network stack. It is capable of fooling tools like *nmap* or *xprobe* into identifying it as any chosen operating system. In addition, it is capable of fooling an attacker into thinking

there are many more servers than there are, by taking additional IP addresses in the network and running services emulating many different vulnerable services. Development seems to have effectively stopped as the last version was released in 2007[15] and the latest commit on the git master branch is from 2013[16]. As such some components, like its OS fingerprint database *nmap-os-db*, are noticeably out of date. Still, it remains in active use.

HoneyPLC brings some of these ideas together. Its purpose is to impersonate an Industrial Control System. In order to do that, it combines multiple different tools to automatically collect the necessary information and then serve a copy of the original ICS with enough interactivity to collect malware from attacks. Its network stack impersonation capabilities are based on Honeyd's. The crawling of the original ICSs web interface is delegated to *wget*[17], and the copy of this web interface is served with *lighttpd*[18].

3.2 Limitations

Both HoneyPLC and SNARE allow the user to crawl a website. HoneyPLC uses wget's recursive download feature[19] for its crawling, while SNARE implements its own crawler based on *aiohttp*[12]. The crawling works very much the same in both cases: Given a starting URL, the crawler requests that URL from the server, extracts the links/URLs from the response, then requests those as well, extracts the links from those responses, and so on. This continues until there are no more unrequested links to follow, or until a user-specified limit is reached. The bodies of the responses are saved as files in a filesystem directory structure mirroring the path structure of the URLs. For example, when requesting `example.com/index.html` the response body would be saved in `<output directory>/index.html`. The created directory is then served, similar to a normal webpage.

The implicit assumption of this approach is that every web resource can be treated as static. This works well for some pages and causes problems for others. For instance, the headers sent by most honeypots described don't match those of the original device at all. Not every HTML file has an explicit encoding, and when the honeypot server implementation guesses, it sometimes guesses wrong. This can lead to obvious mistakes like umlauts being displayed wrong, which makes such a honeypot easy to spot.

But even if the page is displayed correctly, one look at the headers is often enough to identify a difference from the original device, because simple things like the order of the headers are wrong. Newer versions of SNARE save some of the headers encountered by the crawler[20]. The headers it ignores are context information like `content-encoding`, `date` or `cache-control`.

Another problem is that many resources are simply missed: If a resource requires some form of authentication to access, it will not be part of the honeypot, because none of the tools mentioned has support for handling logins. This is unfortunate, as "interesting" interfaces or data are usually behind some sort of authentication. So, while these

honeypots can still tell us that an attack is happening, we will not learn much beyond that, as the attacker cannot attempt to access anything.

Also, any kind of resource fetched dynamically via JavaScript is missed as well, because JavaScript is not executed, so any of those fetches simply do not happen, again leading to easily identifiable honeypots because pages that should work like normal are broken.

CHAPTER 4

Honeypot Creation

To overcome the limitations laid out, we propose the following approach:

First, instead of only saving the body of the response, we record the entire response, and also the entire request which triggered the response. On the one hand, this preserves any relevant header information, like the encoding. On the other, this allows the approach to not only work for HTTP but any stateless protocol.

Second, we do not send just one request per URL, but multiple. The idea is to record responses for different situations, which enables us to handle login scenarios and some other situations with dynamic components. For instance, we take care to crawl every page once with an authenticated session, and once without. For login pages, we make sure to record multiple failed logins as well. CHAMELEON does something similar, but the purpose there is to identify the differences in the responses.

Third, we update some information. This means things like the `Date` header, which should reflect the current timestamp. In the current state, we do not update the message bodies, unlike CHAMELEON, but simple find-and-replace substitutions are considered future work.

Fourth, while serving, we match incoming requests against our records and identify the best fit. Depending on the protocol or the situation, the criteria for the best fit can vary. For HTTP matching the request line first and strictly, and then matching the headers more loosely seems a good heuristic, but for another protocol, something like the smallest Hamming distance might be better.

Fifth, we send back our record of the response to the closest-matching response, updated accordingly. While there is a risk that this response might not match the attacker's expectations, we can at least be sure that we are sending back a response that matches exactly what the original device would have responded to a request similar to the one our honeypot received.

This approach has some nice benefits, apart from the byte-correct responses to many requests: For instance, our honeypot does not support any actual login handling – which would be quite difficult anyway because we do not know how the original service works – and yet it behaves as if it did. If the attacker sends a login request, it will be matched as such, and a response sent back that matches the behaviour of the login mechanism. It will even seemingly refuse wrong login information. And once the attacker has the necessary authentication, their requests will also match the requests that an authenticated user would make. We unwittingly guide the attacker onto a path which our crawler has explored, making the interaction seem genuine.

Of course, this is not perfect and has a chance to go wrong, if an incoming request with wrong login information still happens to match closest to a recorded one with right information, or vice versa. Depending on the situation, this might be rather inconspicuous in some cases, like if the login page seemingly accepts some wrong login information there might just conceivably be an account with these login credentials. But in other situations, this would be rather suspicious, if a page suddenly seems accessible without login. One way to mitigate this is to make sure that unauthorized accesses are well-represented in the original crawl, another way would be to implement some custom request matching which follows the logic of the used login mechanism more closely.

CHAPTER 5

Implementation

5.1 Earlier attempts

Our first approach was to try to follow SNARE's example and use an off-the-shelf Python crawling framework like *Scrapy* or an HTTP client library like *urllib*, *httplib*, *requests*, *grequests*, *requests-threads*, *requests-futures*, *asks* or *aiohttp*. While they differ in features and speed, there is a problem (for our purposes) they all share: They offer raw access to the body of an HTTP response, but not to the other parts, like the status line or the headers. Instead, those are presented in a parsed form, like a key-value map for response headers. This is a problem because the HTTP standard allows for quite a bit of wiggle room in what it considers a valid message, and actual implementations are usually even more lenient. So idiosyncracies of a device, like the order of headers, unnecessary whitespace or lowercase headers, are lost and cannot be consistently recovered from the available information. [1] This makes it impossible to convincingly impersonate any but the simplest, most well-behaved devices. The problem can be alleviated by modifying the code of those libraries to include an unaltered copy of the pre-body section, which we successfully did for *urllib*.

But this does not solve another more fundamental problem: The responses are not interpreted in any meaningful way. What this means in practice is that if a page contains JavaScript to fetch some data dynamically, this fetch simply never happens because the JavaScript is never executed. While JavaScript engines exist and could be used, the much

[1] aiohttp is a special case here. It does offer the `raw_headers` field `https://docs.aiohttp.org/en/stable/client_reference.html#aiohttp.ClientResponse.raw_headers` since around version 4 `https://github.com/aio-libs/aiohttp/commit/7cf0339103ac5e7bdf713563a68068621e7d9fd5`. `raw_headers` is a list of tuples of the key and value of each header, in the order they are read, and only slightly modified. This would probably be enough to reconstruct the headers as sent, but the other problems described persist so it does not change anything.

more practical solution is to just use a browser instead. This also means we don't have to crawl resources like images explicitly, because a browser will just request them while constructing a page for the user.

There are two ways to go about this: One way is to use an embedded browser like *PhantomJS*[21], *CasperJS*[22] or *Splash*[23]. Unfortunately, both CasperJS and PhantomJS have been unmaintained since 2018[24]. Splash remains a viable option. The developers of PhantomJS recommend using *Selenium* instead, which is the other way. Selenium, or more specifically *Selenium WebDriver*, allows us to control a normal desktop browser from a Python program. However, using Splash or Selenium has the same problem as the libraries before: It does not give us access to the raw pre-body section of responses, and this time patching a handful of Python functions is not going to cut it. *selenium-wire*[25] gets close, but also only exposes headers as a dictionary with lenient parsing. **scrapy-selenium**[26] provides a way to have Scrapy spiders use Selenium for actual requests, but might also be unmaintained at this point in time.

So, a different approach to recording the raw HTTP messages is needed. For one, because it is simply not a feature in existing libraries, but also because it would be very inflexible: Because we want to be able to use different crawlers for different situations – maybe a certain crawling library is being blocked by the device, maybe we want to record a different protocol, maybe our crawler simply missed functionality for a certain scenario – relying on the crawler to be able to provide raw access to the network traffic might require major reengineering for every new situation. But if we record the traffic in another place entirely, our crawlers can be as simple or as complex as we want and everything still works.

To record traffic at the operating system level, many tools like *tpcdump* exist. For a clear text protocol, it would be possible to use such a tool to dump all the network traffic, then parse that and create some sort of representation that allows a server implementation to behave accordingly. For our purposes, however, this will not do because of the existence of HTTPS. To decrypt the traffic would require access to the private key of the server, which is not an assumption we can make, especially since we want to have a solution that works without manual intervention. Thankfully, there is yet another way.

By putting a proxy – which we control – between the crawler and the server, we can circumvent the problem by splitting the encrypted connection into two parts: The proxy communicates with the server like a client, so it can decrypt the traffic on its end, and then pretends to be the server to the crawler, re-encrypting the traffic with its certificate so the crawler still sees it as an HTTPS connection. Of course, the certificate does not match that of the server, so we need to install a compatible certificate on the crawler, but since both the proxy and the crawler are under our control, this is not a problem. The next problem is how to make sure that the traffic passes through the proxy since internet packets can be routed in many ways. The easiest solution would be to use the proxy as an HTTP proxy and just have the crawler respect that. HTTP proxies are so common that most or all of the possible crawlers described before would have this feature out of the box. Unfortunately, a request sent to a proxy is composed differently than a

request sent directly to a server. The differences are manageable, but there is a better solution: If we again work at a lower level and just let the crawler believe that the proxy is the server it is supposed to interact with, we can use the recorded traffic as-is. For this, we had a look at *proxy.py*[27], *mitmproxy*[28] and *bettercap*[29], eventually settling on mitmproxy for a mature, well-documented tool that seems to fit the use case best. In addition, it has nice features like automatically generating the certificates we need to install in the crawler, and support for server-side replay[30], which means we don't need to implement a custom server, and can just reuse mitmproxy's functionality instead.

One other thing we wanted to achieve is automatic login. There is a Python package by the name of *autologin*, which promises to do just that. It is based on *formasaurus*, which uses machine learning to identify forms on web pages which can then be filled and submitted. While the idea seems great, the actual implementation is held back by code rot. The last release of autologin on PyPi was in 2017[31] and the last commit on Github in 2018[32], and for formasaurus it's 2018[33] and 2020[34]. Unfortunately, formasaurus is built upon libraries that have seen drastic changes in the last years and has not kept up with those changes. *sklearn* moved the location of the *joblib* module, meaning formasaurus fails to import it correctly. The last commit on GitHub addresses this, but as of September 2022, those changes have not been published to PyPi. For *Flask*, the situation is even worse: Formasaurus depends on exactly version 0.10.1 of Flask, but Flask itself only specifies minimum versions for its dependencies, like *Jinja*. Jinja also moved some modules around, notably the *Markup* module, which again breaks imports. Neither the newest nor the oldest allowed version worked, but by trial and error, we eventually found a working version of Jinja. We repeated this trial and error for a handful of other dependencies. But then we got to *flask-sqlalchemy*. It requires version 0.7 or higher of *sqlalchemy*. Versions 0.7 and 0.8 are both incompatible with *Python 3* and using *Python 2* — which has been retired and does not even receive security support since 2020, so really should not be used for anything anymore — breaks other parts of the project. And any version from 0.9 onwards breaks because of another incompatibility error that we were unable to solve. In summary, we were not able to get autologin to work in any configuration, so we decided to instead implement a simple heuristic that should cover most normal login situations[35].

5.2 Composition of the tool

To impersonate a service as described in Section 4, we need two components: One, a crawler to interact with the original device and store a representation of the traffic. And two, a server that can then use this representation to serve as our actual honeypot. For technical reasons described in Section 5.1, the actual architecture of the crawler is more complicated, and described here:

The first building block is **Docker**[36], together with **Docker Compose**[37]. Docker gives us a layer of abstraction between the host system and our traffic interception. This means we can make sure the traffic of the crawling component passes through the proxy

without changing the host's firewall rules (beyond what Docker does automatically). This could be achieved by other means as well, but the ability to use prebuilt containers simplifies deployment and maintenance drastically, and Compose allows us to accomplish this with a single, declarative file.

The second building block is **mitmproxy**[28]. Not only is it one of the most popular proxy solutions, but it also comes with features that simplify the implementation of both the crawler and the server:

For the crawler we use mitmproxy's *reverse proxy mode*, which means that mitmproxy will impersonate a given URL by relaying every request it receives to the actual server, and the responses back to the client. For this to work with HTTPS, which most websites support these days — sometimes exclusively —, it generates certificates that we install in the client. The traffic that then passes through the proxy is recorded to a file.

For the server, we use mitmproxy's *server-side replay*[30] feature. When this feature is active, mitmproxy matches incoming requests against requests contained in the file we created during the crawling. There are ways to customize this process with options[38]: For our purposes, we activate *response refreshing*[39], which updates the "date", "expires" and "last-modified" headers, as well as cookie expiry times, to match what a running server would send at that moment.
We use *server_replay_nopop*[40], because otherwise, mitmproxy would remove used flows, meaning it would not respond to requests it has seen before.
server_replay_kill_extra[41] kills the connection of any request for which no match can be found.
We also use *server_replay_ignore_host*[42] so that the deployment URL doesn't have to match the URL of the original device.
server_replay_use_headers[43] is used to match on headers. In our case, this means the`Cookie` header, so that logins can be modeled correctly.
And finally, we use *server_replay_ignore_params*[44] to allow some JavaScript fetches that would not work otherwise because of mismatched parameters.

We use **Selenium**[45] as the third building block. It allows us to crawl with an actual browser so we can include requests made by JavaScript. Because setting up and running a browser and the corresponding webdriver in a Docker container comes with some difficulties and would require some maintenance over time, we instead chose to use the official Selenium Standalone Firefox container[46]. Because of our Docker Compose setup this went smoothly, we only had to include a short delay in the crawler so the crawler would not try to begin before the Firefox container was ready to receive orders.

Since the original reasons for discounting all existing Python crawling solutions were solved by introducing a proxy, we are free to use them. We considered using Scrapy, together with scrapy-selenium[26], but it turned to be simpler to just implement the crawler with Selenium as the only dependency besides Python's standard library.

The Compose file for the crawler is listed below. It shows all four involved containers, the volume we use to get the certificates from mitmproxy to the crawler and the network

configuration we use to route the relevant traffic to the proxy (note the `extra_hosts` configuration of the containers).

```
services:
  proxy:
    image: mitmproxy/mitmproxy
    command: ["mitmdump",
            "-w", "/home/mitmproxy/out/${DOMAIN}.mitm",
            "--mode", "reverse:${SPEC}",
            "--listen-port", "${PORT}", "--quiet"]
    tty: true
    ports:
      - "8080:8080"
      - "8081:8081"
    networks:
      internal:
        ipv4_address: 172.28.0.2
    volumes:
      - ca-certs:/home/mitmproxy/.mitmproxy
      - ../out:/home/mitmproxy/out
    extra_hosts:
      - "${DOMAIN}:${UPSTREAM_IP}"

  crawl:
    build: ./crawl
    command: ["python", "/usr/src/crawl/crawl.py",
            "${STARTING_URL}", "${LOGIN_URL}",
            "${LOGIN_A}", "${LOGIN_B}"]
    networks:
      internal:
        ipv4_address: 172.28.0.3
    extra_hosts:
      - "${DOMAIN}:172.28.0.2"
    volumes:
      - ca-certs:/ca-certs
    depends_on:
      - proxy
      - anon_browser
      - login_browser

  anon_browser:
    image: selenium/standalone-chrome
    environment:
```

```
      - SE_OPTS=--log-level SEVERE
    networks:
      internal:
        ipv4_address: 172.28.0.4
    extra_hosts:
      - "${DOMAIN}:172.28.0.2"
    depends_on:
      - proxy
    shm_size: 2g # crashes otherwise

  login_browser:
    image: selenium/standalone-chrome
    environment:
      - SE_OPTS=--log-level SEVERE
    networks:
      internal:
        ipv4_address: 172.28.0.5
    extra_hosts:
      - "${DOMAIN}:172.28.0.2"
    depends_on:
      - proxy
    shm_size: 2g

networks:
  internal:
    ipam:
      config:
        - subnet: 172.28.0.0/16

volumes:
  ca-certs:
```

The server component is just mitmproxy again, using its *server side replay* functionality[30], with a single command like this:

```
sudo mitmproxy --listen-host 0.0.0.0 --listen-port 80
        --server-replay ../out/res-dev0.test.mitm
        --server-replay-nopop
        --server-replay-refresh
        --server-replay-kill-extra
        --set server_replay_ignore_host
        --set server_replay_use_headers=Cookie
```

5.3 Limitations and future work

SyrupPot, our tool, has limitations. Some of these are fundamental limitations of the approach, and others are limitations of the implementation. Of these, some could be alleviated with additional engineering effort, but some are inherent to low-interaction honeypots in general.

The first problem is how to provoke all (relevant) behaviour. We recursively crawl a website and also execute JavaScript to find as many URLs as we can, but there could always be pages not linked to from our starting page, not even indirectly. Administration interfaces are often not linked in such a way. A solution would be to have a mechanism for adding additional URLs to crawl, but that requires knowledge of those sites. Fuzzing URLs — like has been done by Kim, Mingeun, et al.[47] — might lead to some success, but we are faced with practically unbounded search space[48]. Allowing the crawler to leave the original domain a bit might help in some cases, in case links to otherwise unconnected pages exist in a related domain. A good compromise might be to have a list of paths that often exist outside the main website structure, like `robots.txt` or `wp-admin`. Another way would be to allow potentially state-changing request methods like POST, but probably only in combination with a rigorous blacklist, so as not to damage the original service.

The second problem is that because we know nothing of the internal workings of the original service, we cannot reproduce its internal complexity. Let's assume a simple webpage on which a user can type in a number and the server responds with the square of that number. For our honeypot to be able to correctly respond to any request an attacker might send, we have to crawl absolutely every number the server accepts. Even if our crawler could enumerate all those possible inputs, which it cannot, this would be completely infeasible. For this simple example, an AI approach might do better, but for more complex cases, involving cryptography or randomness or some secret context information, that would fail as well. The only real solution — apart from running a high-interaction honeypot — is to manually or semi-automatically identify such scenarios and implement some handling logic, as was done for CHAMELEON.

The third and final problem is that of maintaining state: SyrupPot's server treats every interaction as purely stateless. Because of this, our honeypot scales nicely to any number of connections, because one can always simply spawn more independent instances. Another way to look at it is to say that no data is ever written by the server, only read. The downside of this is that actual HTTP servers often keep quite a bit of state, whether this means using session cookies or allowing file uploads or any number of things. Our honeypot can only handle login scenarios because the process is baked into the data on which our server operates. This is a severe limitation, but solving it by maintaining state would mean manual intervention and giving up some of the scaling properties.

Apart from solving the outlined problems, there is more work to be done: We implemented SyrupPot using only Chrome as a crawler, so we don't cover additional user agents. While it is unlikely, an IoT device could respond differently depending on the received user agent.

There are two simple ways out of this though: Because of the way we structured our tool, it's enough to add more Selenium containers (including the `extra_hosts` entry), include them in the crawling script, and maybe add them to the headers mitmproxy considers for matching requests. The other way would be to simply tell Chrome to use a different user agent every time. Covering more user agents means sending a lot more requests though, which means the crawling takes longer. For reasons of usability, we recommend only doing this if and when necessary.

Another improvement that could be made would be to try to resist identification as done by Vetterl and Clayton[3]. It might be possible to harden our honeypot against such an identification attack by including identification probes during the crawling phase, which would then be part of our honeypot's repertoire.

And finally, we would like to expand SyrupPot to cover other protocols besides HTTP. Any interceptable stateless protocol should work, like Gemini[49], DNS[50] or WHOIS[51]. Doing this would require implementing crawlers for these protocols, which might prove problematic, especially for protocols which don't have explicit links to follow.

CHAPTER 6

Evaluation

To evaluate the effectiveness of SyrupPot, we used it on a few different devices and planned to compare the resulting honeypots to the original devices and to honeypots created by SNARE.

We had the following routers at our disposal:

- tp-link TL-WR841N Version 14.0
- Tenda AC1200
- PLANET WNRT-617
- NETGEAR AC1900 (R7000)

In a preliminary examination, we learned that both the PLANET and the NETGEAR routers do not use a login page, instead relying on HTTP authentication[52], which is, at the time of writing, not implemented in our crawler. It could be done, but because SNARE is unable to clone any page requiring authentication anyway, and all pages on these routers require authentication, we still would not be able to compare the tools. That leaves us with the tp-link and the Tenda routers.

The tp-link router sports a login page requiring username and password, and the actual login logic is handled in JavaScript. By default, the login information is “admin/admin”, but of course, this can be changed in the settings. The Tenda router’s login page only has a password field, and the password is set during the first setup. The computer we used for the evaluation is an Acer Aspire VN7-592G running an up-to-date version of Manjaro Linux with kernel version 6.0.0-1-MANJARO (64-bit).

Having used SNARE before with moderate success, we intended to create honeypots from the two workable routers with SNARE and SyrupPot, and then compare the resulting honeypots. But, to our surprise, we ran into problems trying to use SNARE:

6.1 SNARE

We considered two versions of SNARE: Version 0.3, released in 2018[53], and the latest development version on GitHub, with the last commit being from 2021[54]. Originally we planned to use version 0.3, since it is the latest proper release. It was "tested primarily with >=3.4"[55], and the cloner seemed to work well with Python 3.10, finishing within seconds. The honeypot however, failed to start because of dependency problems. So we tried older Python versions, which came with its own set of issues because everything older than Python 3.7 is not supported anymore. But this only led to other problems which we were unable to solve.

Instead we tried to run the latest development version, which requires Python 3.6. Given its newer codebase and updated dependencies this seemed promising, but it ran into an import error on Python 3.10. This is a known problem caused by reorganised modules[56].

Because trying to run old, unsupported Python versions on Manjaro in 2022 had turned out to be such a problem, we decided to try to match the original system used to test SNARE as closely as we could. For this we set up a Virtual Machine (VM) running Ubuntu 18.04 LTS. This still did not fix the issues plaguing SNARE version 0.3, but thankfully, it did seem to work for the latest development version. The cloner, again, ran without problems, and this time, so did the honeypot. But to our astonishment, it turned out to be unusable: The honeypot would not respond to any request, while also not showing any error messages.

We suspect that SNARE cannot successfully impersonate devices which don't have a public domain and are only reachable by IP, as this was a problem we also had to overcome with our implementation, but that is purely speculation. Maybe the JavaScript-based and direction-heavy ways the router's interfaces work turned out to be a problem. It could also conceivably have been an error on our part. All we can say for certain is that we were unable to make working honeypots with SNARE. As such, we had to finish our evaluation without it, instead only comparing the honeypots our tool created to the original devices.

6.2 SyrupPot

We used the following commands to create and deploy honeypots based on the two suitable routers:

Copy Tenda router `sudo sh syruppot.sh 192.168.0.1 http://tenda.test/index.html http://tenda.test/login.html testtest`

Deploy Tenda honeypot `sudo mitmproxy --listen-host 0.0.0.0 --listen-port 80 --server-replay ../out/tenda.test.mitm --server-replay-nopop --server-replay-refresh --server-replay-kill-extra --set server_replay_ignore_host`

Copy tp-link router `sudo sh syruppot.sh 192.168.0.1 http://res-dev0.test/ http://res-dev0.test/ admin admin`

Deploy tp-link honeypot
```
sudo mitmproxy --listen-host 0.0.0.0
--listen-port 80 --server-replay ../out/res-dev0.test.mitm
--server-replay-nopop --server-replay-refresh
--server-replay-kill-extra --set server_replay_ignore_host
--set server_replay_use_headers=Cookie
--set server_replay_ignore_params=_
```

The URLs `tenda.test` and `res-dev0.test` are made up and only used so the traffic can by directed via DNS entries. In most cases they can be ignored because we use *server_replay_ignore_host* while deploying the copies, but they can be important if one wants to not use server_replay_ignore_host or if the service uses absolute links to its own pages. In this case one should use an URL at which the honeypot will then be reachable.

Also of note is that we use slightly different commands for our honeypot deployments. We left out *server_replay_use_headers=Cookie* for the Tenda router because it uses complex cookies which could not be matched against the cookies on record. And we added *server_replay_ignore_params=_* for the tp-link router because it uses the "_" parameters with changing values, but mitmproxy by default matches those parameters strictly. The fixes for both problems are simple enough but would require creating a mitmproxy plugin.

When interacting with our honeypots, they mostly met our expectations. The login pages worked like real login pages on both honeypots: With the right login information, the user is redirected to the main interface page, and with wrong login information, they are not. The actual interfaces behind the logins are a bit more of a mixed bag. They look like the interfaces they are copying, but some elements are either not interactable or downright missing. The reason for this is that some parts of these pages are requested with randomly generated numbers as queries, and mitmproxy unfortunately lacks a feature to ignore they query string wholesale. Again, this could be easily implemented in a plugin.

Looking at the structure of the responses, and the headers in particular, they are identical from those of the original devices, apart from the `Date` and `Last-modified` headers, which only the Tenda router uses. This comes as no suprise given that those responses are just replayed from the records, except for the updated fields. Those `Date` and `Last-modified` fields are noteworthy however. The Tenda router had sent responded with `Date: Thu Jan 01 03:39:44 1970`, and our mitmproxy honeypot had updated it to `Date: Sun, 04 Jan 1970 21:19:05 GMT`. The change in time is correct and wanted — even if the actual date is obviously nonsense — but the format has also changed slightly.

Looking at RFC 7231[57], we can see that the Tenda router breaks protocol twice: On one hand, it's not allowed to even send the `Date` header at all because it doesn't have the right time: "An origin server MUST NOT send a Date header field if it does not have a clock capable of providing a reasonable approximation of the current instance in Coordinated Universal Time."[58] And on the other hand, one look at the provided form example "`Date: Tue, 15 Nov 1994 08:12:31 GMT`"[58] shows us that the router uses the wrong format, which mitmproxy cleans up in the process of updating it. The same is true for the `Last-modified` header. In this particular case, it would be better to use *–no-server-replay-refresh*[59] and just leave the headers as they are, since the timestamp they provide is meaningless anyway, or handle it in a plugin.

Two flaws remains: Because we deactivated Cookie matching for our Tenda honeypot, the actual interface can be accessed without having logged in successfully. For our demonstration this does not really matter, but for a proper honeypot Cookies should be handled accordingly. And while login attempts with wrong information do not work, the response they receive does not always match what the original device would answer because there is no matching traffic on record. This too could be solved in a mitmproxy plugin, by simply matching any wrong login attempts to one of the wrong logins attempts on record.

We feel like this sufficiently demonstrates the usefulness and potential of our approach and hope it will lead to smarter, more effective honeypots in the future.

CHAPTER 7

Conclusion

In this paper, we laid out the available tools for creating low-interaction honeypots automatically and their shortcomings. To overcome these shortcomings, we suggested an approach that looks at request/response pairs as a whole, in order to allow a honeypot to be more convincing in most cases. It also enables such a honeypot to handle additional challenges not covered by other approaches. We implemented a tool which can create low-interaction HTTP honeypots without specialised knowledge or dynamic user input. We describe the engineering challenges we faced, some of which we were able to solve. For those problems we could overcome we described the architectural changes we had to make in order to make the implementation possible. And for those we did not solve, we explained the problems to the best of our ability, and what could be done to solve them. Finally, we demonstrated the approach's potential in practical examples with our tool. The resulting work not only addresses the limitations of existing approaches but can also serve as a basis for better honeypots in the future.

Acronyms

IoT Internet of Things. 1

VM Virtual Machine. 22

Bibliography

[1] https://cybersecurity.att.com/blogs/labs-research/shikitega-new-stealthy-malware-targeting-linux.

[2] I. Mokube and M. Adams, "Honeypots: concepts, approaches, and challenges," in *Proceedings of the 45th annual southeast regional conference*, pp. 321–326, 2007.

[3] A. Vetterl and R. Clayton, "Bitter harvest: Systematically fingerprinting low-and medium-interaction honeypots at internet scale," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[4] https://honeyscore.shodan.io/.

[5] E. López-Morales, C. Rubio-Medrano, A. Doupé, Y. Shoshitaishvili, R. Wang, T. Bao, and G.-J. Ahn, "Honeyplc: A next-generation honeypot for industrial control systems," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 279–291, 2020.

[6] https://docs.mitmproxy.org/stable/api/mitmproxy/flow.html.

[7] http://conpot.org/.

[8] https://www.honeynet.org/projects/active/dionaea/.

[9] http://mushmush.org/.

[10] https://github.com/mushorg/snare.

[11] https://github.com/mushorg/tanner.

[12] https://docs.aiohttp.org/en/stable/.

[13] M. Musch, M. Härterich, and M. Johns, "Towards an automatic generation of low-interaction web application honeypots," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pp. 1–6, 2018.

[14] N. Provos *et al.*, "A virtual honeypot framework.," in *USENIX Security Symposium*, vol. 173, pp. 1–14, 2004.

[15] http://www.honeyd.org/.

[16] https://github.com/DataSoft/Honeyd.

[17] https://www.gnu.org/software/wget/.

[18] https://www.lighttpd.net/.

[19] https://www.gnu.org/software/wget/manual/html_node/Recursive-Download.html.

[20] https://github.com/mushorg/snare/commit/098298daacb75d403a26f88d516220d9d00894d2.

[21] https://phantomjs.org/.

[22] https://github.com/casperjs/casperjs.

[23] https://github.com/scrapinghub/splash.

[24] https://github.com/ariya/phantomjs/issues/15344.

[25] https://github.com/wkeeling/selenium-wire.

[26] https://github.com/clemfromspace/scrapy-selenium.

[27] https://github.com/abhinavsingh/proxy.py.

[28] https://mitmproxy.org/.

[29] https://www.bettercap.org/.

[30] https://docs.mitmproxy.org/stable/overview-features/#server-side-replay.

[31] https://pypi.org/project/autologin/.

[32] https://github.com/TeamHG-Memex/autologin.

[33] https://pypi.org/project/formasaurus/.

[34] https://github.com/TeamHG-Memex/Formasaurus.

[35] T. of the Northwest, "How browser's identify login forms?." https://stackoverflow.com/a/1976796.

[36] https://www.docker.com/.

[37] https://docs.docker.com/compose/.

[38] https://docs.mitmproxy.org/stable/concepts-options/#server_replay.

[39] https://docs.mitmproxy.org/stable/overview-features/#response-refreshing.

[40] https://docs.mitmproxy.org/stable/concepts-options/#server_replay_nopop.

[41] https://docs.mitmproxy.org/stable/concepts-options/#server_replay_kill_extra.

[42] https://docs.mitmproxy.org/stable/concepts-options/#server_replay_ignore_host.

[43] https://docs.mitmproxy.org/stable/concepts-options/#server_replay_use_headers.

[44] https://docs.mitmproxy.org/stable/concepts-options/#server_replay_ignore_params.

[45] https://www.selenium.dev/.

[46] https://hub.docker.com/r/selenium/standalone-firefox.

[47] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Annual Computer Security Applications Conference*, pp. 733–745, 2020.

[48] https://stackoverflow.com/questions/417142/what-is-the-maximum-length-of-a-url-in-different-browsers.

[49] https://gemini.circumlunar.space/.

[50] https://en.wikipedia.org/wiki/Domain_Name_System.

[51] https://en.wikipedia.org/wiki/WHOIS.

[52] https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication.

[53] https://github.com/mushorg/snare/releases/tag/v0.3.

[54] https://github.com/mushorg/snare/commit/0919a80838eb0823a3b7029b0264628ee0a36211.

[55] https://github.com/mushorg/snare/tree/0919a80838eb0823a3b7029b0264628ee0a3621

[56] https://github.com/mushorg/snare/issues/315.

[57] https://www.rfc-editor.org/rfc/rfc7231.

[58] https://www.rfc-editor.org/rfc/rfc7231#section-7.1.1.2.

[59] https://github.com/mushorg/snare/issues/315.