b
UNIVERSITÄT
BERN

Investigating Phishing on Demand

Bachelor Thesis

Pascal Gerig

from

Sachseln OW, Switzerland

Faculty of Science
University of Bern

30 April 2020

Prof. Dr. Oscar Nierstrasz

Pascal Gadient

Software Composition Group

Institute of Computer Science

University of Bern, Switzerland

Abstract

Gathering protected information by disguising an attacker as a trustworthy contact in electronic communication, also known as “phishing,” is the primary technique attackers use to steal sensitive data. Phishing websites are mainly static and barely synchronize with the original website.

We investigate “Phishing on Demand”, a technique to dynamically replicate any website for phishing purposes. The replicas are available with a few clicks and are always in sync with the original web pages.

Our studies with a proof of concept show that this phishing technique is highly effective. For instance, we could successfully run phishing campaigns against major Swiss e-banking websites with two-factor authentication.

With this thesis, we show that there is a demand for more robust visual similarity algorithms for websites that are able to track changes applied to original sites such as insertions of banners, rewritings of text, or alterations to graphics.

Contents

1 Introduction **1**

2 Background **3**

2.1 Terminology . . . 3

2.2 Typical Phishing Scenario . . . 4

2.2.1 Email . . . 4

2.2.2 Landing site . . . 4

2.2.3 Monetization . . . 4

3 Related Work **6**

3.1 Modus Operandi . . . 6

3.2 Anti-phishing Measures . . . 7

4 Phishing on Demand **9**

4.1 Process . . . 9

4.2 Advantages . . . 10

4.2.1 Accuracy . . . 11

4.2.2 Back-end Logic . . . 11

4.2.3 Browser Plug-in Support . . . 11

4.2.4 Completeness . . . 12

4.2.5 Collected Data . . . 12

4.2.6 Continuous Tracking . . . 12

4.2.7 Recency . . . 12

4.2.8 Required Knowledge . . . 13

4.2.9 Required Set-up Time . . . 13

5 Proof of Concept **14**

5.1 Implementation Details . . . 15

5.1.1 Client side Features . . . 16

5.1.1.1 HTML Title and Favicon . . . 16

5.1.1.2 Rendering of Hyperlinks . . . 16

5.1.1.3 Rendering of Screenshots . . . 17
5.1.1.4 Rendering of Input boxes . . . 18
5.1.1.5 Replication of URL Subpaths . . . 19
5.1.1.6 User Tracking . . . 20
5.1.2 Server side Features . . . 23
5.1.2.1 Consistent Image Order . . . 23
5.1.2.2 Logging . . . 24
5.1.2.3 Oversize Screenshotting . . . 24
5.1.2.4 Tab Support . . . 24
5.1.2.5 Firefox Plug-in Support . . . 25
5.2 Operation . . . 25

6 Empirical Study **27**
6.1 Questionnaire . . . 27
6.2 Performing the tasks . . . 28
6.3 Findings . . . 28
6.3.1 Usability . . . 29
6.3.2 Authenticity . . . 29
6.3.3 Further Observations . . . 30

7 Limitations, Mitigations and Opportunities **31**
7.1 Limitations . . . 31
7.1.1 Computational Delays . . . 31
7.1.2 Domain Use . . . 32
7.1.3 Transmission Delays . . . 32
7.1.4 Limited Interactivity . . . 32
7.1.5 Phishing Campaigns . . . 33
7.2 Mitigation Strategies . . . 33
7.2.1 Flow Analysis . . . 33
7.2.2 Domain Analysis . . . 33
7.3 Opportunities . . . 34
7.3.1 Improving Browsing Security . . . 34
7.3.2 Bypassing Filtering of HTTP Content . . . 34

8 Threats to Validity **35**

9 Conclusion **36**

A Anleitung zum wissenschaftlichen Arbeiten **39**

1 Introduction

Phishing is a social-engineering technique to collect sensitive information from people without their knowledge and consent. It often deceives a victim to land on a fake website (*i.e.*, phishing website) through which the attacker collects otherwise sensitive data. Phishing websites are not hard to develop, and tools exist that support criminals to implement and launch a phishing campaign without particular IT skills. There are also numerous countermeasures to circumvent exposure to phishing. Nonetheless, it is one of the most common cybercrimes across the globe. For example, 65% of businesses in the U.S. suffered from at least one successful phishing attack in 2019 [19].

Phishing websites are mainly static, not up-to-date, and lack server-side features, which increases the likelihood that victims will identify their exposure to phishing. In other words, the content and template of phishing websites are not necessarily in sync with the original ones, and protected features that need interaction with a server are missing.

In this thesis we introduce "*Phishing on Demand*" (PhD), a new technique inspired by the *browser isolation* technology[1] to develop phishing websites that are, to a great extent, identical to their genuine counterparts. The idea is to provide a phishing framework that acts as a proxy between a client browser and a server. The framework redirects web requests to the server, renders every response, and provides the client with "enriched images" of the response. More precisely, a resulting web page in the client browser presents

[1] `https://www.secjuice.com/remote-browser-isolation-vendors/`

images of the original website that are enriched with interactive UI elements to provide a genuine web experience.

We implemented a proof of concept based on PhD, and compared it with two popular phishing frameworks. We found that PhD is much easier to work with: it requires criminals to only specify the domain names they would like to phish. Thanks to its proxy architecture, PhD captures every single user interaction, traces the victim throughout a browsing session, and stores sensitive inputs. Unlike other frameworks, the template and the content of a phished website are always in sync with the original website. Importantly, PhD is the first phishing framework supporting features that need interaction with the original server: from a "simple query recommendation while typing into a search box" to a "two-factor authentication (2FA) measure." For instance, we could phish the login page of a major Swiss e-banking website that uses 2FA.

We also investigated the extent to which PhD is successful to phish sensitive data in practice. We phished websites such as `gmail.com` and `amazon.de`, and asked seven participants to perform five predefined tasks disguised as website usability studies. The obtained result is promising. From 21 page visits that involved phished websites, only six (*i.e.*, 29%) were considered suspicious.

In summary, phishing campaigns usually aim only to collect credentials by phishing login pages which often have simple UIs and are free from complicated user interactions. PhD not only provides a fast and simple setup of phishing campaigns of that kind, but it also bypasses authentication measures like 2FA. What is more, PhD enables tracing the victim throughout *any* browsing session. This, of course, requires an increased bandwidth and a lower network latency to provide victims with a seemingly genuine browsing experience. We believe this is not a concern with the emerge of fast network interconnects *e.g.*, 5G networks. As a result, we see an increased need of using a variety of anti-phishing techniques in order to mitigate such threats. Hence, we encourage the security community to focus more on the visual aspects of phishing websites, *i.e.*, developing more robust visual similarity algorithms specifically for websites that are able to detect the changes applied to an original site, *e.g.*, in the layout, text, or graphics.

In this work we closely follow ethical principles. In all our experiments we used credentials that we solely created for the sake of phishing. No participant had to provide personal data. Our implementation only supports one user at a time and is unavailable as source code to make our tool unfit for "professional" use.

The remainder of this thesis is structured as follows. We discuss general phishing in chapter 2, present related work in chapter 3 and we elaborate on the concept of PhD in chapter 4, where we also compare existing phishing frameworks. We provide implementation details and remarks for PhD's operation in chapter 5. Next, we investigate the user feedback for the proof of concept's phished sites in chapter 6 and we elaborate on the limitations, mitigations, and opportunities in chapter 7. Finally, we report the threats to validity in chapter 8 and conclude in chapter 9.

2

Background

In this chapter, we briefly elaborate on phishing to ease the understanding of this work. We first explain basic phishing terms, before we present a typical phishing scenario that could be adapted for PhD.

2.1 Terminology

The most common form of phishing, which does not address individual recipients, is called *bulk phishing*. The term "bulk" refers to the binge-processing of email addresses in a list of potential victims that the adversary obtained through underground markets. In such cases, the adversary feeds the list into a mass mailing software which then automatically addresses an email to each recipient in the list. More sophisticated *spear-phishing* attacks exploit knowledge about a victim and the environment, *e.g.*, from social networks and the employer. Attackers use that information to build trust before they convince the victim to do tasks at their hand. For instance, an attacker could pretend to be the CEO of the victim's employer and ask the victim to provide credentials unavailable to outsiders. When spear-phishing addresses multiple high-profile targets, *e.g.*, leading politicians, senior executives, or heads of finance, it is called *whaling*. Phishing attempts from one or more scammers that leverage one particular phishing site constitute a phishing *campaign*.

Phishing attacks are typically performed using *phishing kits*, *i.e.*, a set of files that includes a template that mimics the design of the website being impersonated, server side code to capture and send submitted

data to the phisher, and optionally code to filter out unwanted traffic or implement other countermeasures against the anti-phishing community [18].

2.2 Typical Phishing Scenario

According to Hong [9], website phishing usually comprises three major phases: i) the potential victim receives a phishing email, ii) the victim takes action and activates the embedded web URI to land on a phishing website or to download and install a malware that provides more privileges to the attackers, and iii) the criminal collects and monetizes the provided information. The remainder of this section covers each phase in more detail.

2.2.1 Email

The victim receives an email or other electronic message containing an obfuscated URL, *e.g.*, generated by a URL shortening service, or a URL that appears visually similar by using a marginally different Internationalized Domain Name (IDN). Preferably, personal information is used for the salutation, *e.g.*, correct choice of the language, names, and, if required, the postal address. The email must not contain any spelling mistakes, and should closely resemble the original corporate identity.

2.2.2 Landing site

The victim activates the link to the phishing website. Depending on the browser and the IDN, no or minor security warnings are displayed to the victim. After successfully loading the website, the victim enters and submits the requested data.

2.2.3 Monetization

Depending on the phished data, two major monetization strategies exist: direct monetization for data concerning valuable assets (DM), and indirect monetization for any data (IM).

DM is used for credentials related to financial services, and for accounts holding virtual goods, *e.g.*, video game accounts that hold valuable in-game collectables. Phishers sell such assets individually, *e.g.*, by initiating money transfers to foreign bank accounts where the money is immediately withdrawn at ATMs. Although the major video game distributor *Steam* forces two-factor authentication (2FA) by email or mobile app when using their online shop, there still exist numerous providers that do not provide additional security with their default settings.

IM can be used for any data and involves underground marketplaces which host fraudulent activities. Through underground forums, phishers sell less valuable credentials that provide access to services based on a recurring fee, *e.g.*, audio or video streaming services and news websites, or that can be misused for future phishing campaigns, *e.g.*, social networks. Phishers are forced to sell them in bulk due to the low value of such logins. As a result, phishers started to specialize, *i.e.*, they only focus on phishing and

avoid selling individual logins to customers. The people who organize the sale of individual logins to customers are called "purchasers", who, in turn, recruit ordinary people as mules to launder money and virtual goods. These ordinary people are commonly recruited with "work at home" email job offers that yield a small commission. Generally, the purchasers rely on bank accounts that reside in countries with weak law enforcement and work with hard to track currencies, *e.g.*, BitCoin.

In a typical phishing scenario, PhD only focuses on the technical aspects of building a replicated website, *i.e.*, subsection 2.2.2.

3

Related Work

The works presented in section 3.1 provide a state-of-the-art overview of existing techniques used by phishers. The literature we list in section 3.2 motivated the development for our framework.

3.1 Modus Operandi

According to Hong *et al.*, phishing attacks initially targeted general consumers, aiming to steal identity and credit-card information, but evolved to also include more lucrative high-profile targets, aiming to steal intellectual property, corporate secrets, and sensitive information concerning national security [9].

Typically, phishing kits are deployed to servers under direct control of phishers, to free hosting services [16], to hijacked website hosting providers that host one or more legitimate website [7], to public clouds [6], and even to bot-nets [15]. Hosting providers are particularly at stake, because more than 95% of them do not run an anti-virus scan with up-to-date signatures once a month to detect phishing sites [2]. Such a lenient behavior encourages criminals to run privileged escalation attacks that might yield access to other websites published on the same server that could be modified and misused for phishing as well. When a phishing kit that targets a specific website is deployed, the manually applied modifications are rather minor as Cui *et al.* found when they compared the DOM of different phishing sites [5], *e.g.*, adversaries add an input form, replace a few images, or some text. As a covert measure, 95% of the deployed phishing kits are enforcing `htaccess` rules that block "unwanted" visitors as Oest *et al.* discovered when they

examined 1 794 live phishing kits from 2016 through mid-2017 [18].

With their tool "PhishEye", Han *et al.* were able to monitor sandboxed phishing kits and their operators in action [7]. They show that the installation and testing phase usually takes the fraudsters between one and twenty minutes for one deployment. Furthermore, they observed that typically the first victim approaches a phishing website around 48 hours after successful testing, whereas they detected the last victim around two weeks after installation. This is interesting, because Moore *et al.* studied the lifetimes of phishing websites almost ten years earlier and found that they live on average between 62 hours and 1.2 weeks [16], which is much more short-lived. Typical phishing targets are major companies that provide a login to customers based on which they offer services, *e.g.*, Paypal, Apple, Google, Facebook [7].

Adopting a ready to deploy phishing kit also raises threats to its user. Cova *et al.* observed that criminals started to introduce in their kits hidden backdoors that transmit the collected data not only to its user, but also to third parties, *e.g.*, its originator [4]. McCalley *et al.* show that such hidden backdoors have evolved using different email address encoding techniques, *e.g.*, hexadecimal string representations or custom array-related code [14]. Moore *et al.* investigated the re-compromising of internet hosts and found that by performing Google searches using versioning information of phishing kits, attackers discover hijacked and still vulnerable phishing hosts ready to take additional phishing kits [17]. Finally, Birk *et al.* reveal that organized cyber-crime grants custom requests for phishing kits [1], however it is not clear if such individual solutions contain backdoors.

3.2 Anti-phishing Measures

A large body of research has focused on anti-phishing strategies. Hong *et al.* use several lexical features of existing phishing URLs, curated in PhishTank,[1] such as a URL's length or the number of special symbols to build a classifier that detects potential phishing URLs [10]. There exist similar works that use other features such as age of domain in combination with the WHOIS record [11] or the presence of HTTPS [20]. There also exist similar works that use other classification algorithms, *e.g.*, deep neural networks [12], or a combination of them [21]. Fortunately, many of these techniques are available in practice, *e.g.*, by using the "Google Safe Browsing API"[2] or the "Kaspersky Protection Plugin".

Researchers have proposed techniques that detect phishing websites from the content (text, code, rendered images) of websites. For instance, Zhang *et al.* implemented a text-based classifier that extracts, among other data, a website's five most frequent words and sends them to Google to verify if the returned results match the domain in question [22]. Liu *et al.* use the HTML source code to extract the layout of a website to which they apply an image segmentation algorithm [13]. They compare, among other data, the resulting image segments with a manually established ground-truth dataset to provide informed guesses. Chiew *et al.* extract company logo images from a website under test and process them with Google's image search to conclude whether a page is phished [3]. They assume that an image and the corresponding website are

[1] `https://www.phishtank.com/`
[2] `https://safebrowsing.google.com`

legitimate when its URL matches one of the top rated URLs returned by Google. Hara *et al.* compare the screenshot of a website with genuine screenshots that they already have in a database [8].

4

Phishing on Demand

We present "Phishing on Demand," a framework to develop phishing websites that are, to a great extent, identical to their genuine counterparts. This framework acts as a web proxy between a victim's browser and a web server. It renders every web response, and provides the victim with "enriched images" of the requested web page. The framework enables collecting sensitive data, as well as tracing a full browsing session. Furthermore, popular web browsers are supported, *e.g.*, Mozilla Firefox, Microsoft Edge, or Google Chrome. Lastly, working with PhD is very easy, and deploying a phishing website is extremely fast.

In the following we explain the characteristics of the PhD phishing framework, and we discuss its key features. Both the front- and back-end are publicly available on Github.[1]

4.1 Process

The responsibilities of a client and the server are revealed in Figure 4.1. The grey rectangles show the involved devices, *i.e.*, the victim's client computer on the left and the adversary's server on the right. The blue rectangles illustrate the executed applications, *i.e.*, the web browser started by the victim, and the Java server daemon as well as the headless web browser started by the adversary. The arrows indicate the different steps. The steps that describe an action use numbers, whereas transmitted data use letters.

[1] `https://github.com/pascalgerig/PhishingOnDemand`

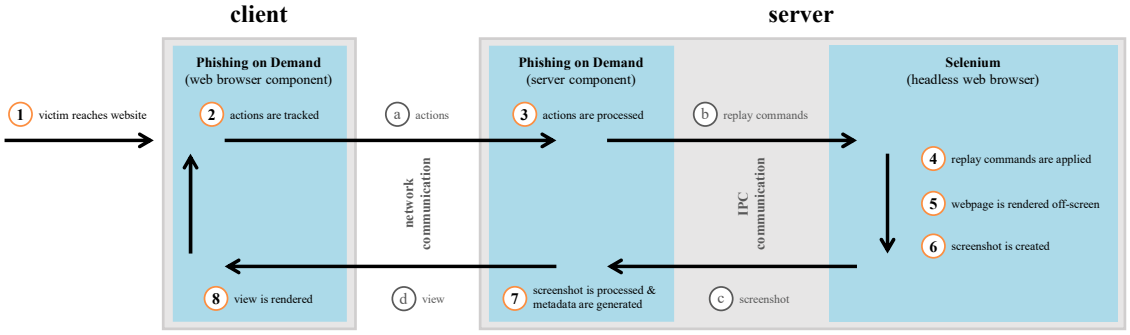

Figure 4.1: Responsibilities of a client and the server

Next, we discuss each step in detail. ① A victim must be tricked into opening a website operated by PhD. This step is identical to existing techniques where attackers convince victims to open a certain URL, for example, by email, instant messaging, or phone. If the victim's browser requests such a URL, the PhD server component quickly spawns a headless browser instance with the pre-configured website, takes a screenshot and sends it back to the client attached to the PhD web browser component which now becomes active on the client. ② The delivered JavaScript code tracks the victim's mouse and keyboard actions and sends them through the internet to the Java-based server daemon. ⓐ The transmitted actions include mouse click coordinates, and keyboard hits. ③ The server daemon decodes the received actions and generates replay commands for the headless web browser following the same order. The commands are sent by using *Inter-Process Communication (IPC)*. ⓑ The forwarded replay commands might look like "execute click at point (52/142)", or "press the keys `J`, `o`, `h`, `n`." ④ The headless web browser maintained by Selenium executes the collected replay actions in the same order they were received. ⑤ The web browser starts to re-render the website based on the provided actions. The rendering itself is by default performed off-screen, *i.e.*, hidden in memory without the use of any visible UI elements, because it enables support for arbitrary large screen resolutions at clients. ⑥ When the configured time-out expires, the headless browser takes a screenshot of the rendered website and sends it back to the server daemon by using IPC. ⓒ The screenshot is encoded in an image format understood by modern browsers. ⑦ The server daemon processes the screenshot and traverses the headless browser's DOM of the previously rendered website to find and extract embedded UI elements. The resulting view is then transmitted over the internet back to the client. ⓓ The view consists of the screenshot, the location and content of text boxes or buttons as well as hyperlinks. ⑧ The JavaScript code executed on the client receives the view, decodes the data, and renders it within the client's browser ensuring a seamless update between each cycle.

4.2 Advantages

Our implementation provides several features unseen in existing implementations. We discuss each feature and explain how traditional template-based, hand-crafted methods, and PhD support them. We rely for information about template-based methods on the two comprehensive open-source projects *Gophish* and

Phishing Frenzy, and for information about hand-crafted methods on our own expertise. Please note that the opportunities offered by handcrafted websites can be partially transferred to template-based methods due to their support for manual code interventions. We expect that traditional phishing pages are created by using these techniques based on the findings of Cova *et al.* who performed a comprehensive study on more than 500 phishing frameworks found in the wild [4]. They mention that phishing kits usually comprise two types of files: the files to display a copy of the targeted website, and the scripts used to save the phished information and send it to criminals.

4.2.1 Accuracy

Accuracy refers to the visual similarity of a phished page compared to the original. If the replica is accurate, no differences can be observed by a potential victim. *Template-based* solutions barely support websites that use dynamic content, *e.g.*, overlays created with JavaScript, because templates cannot automatically adapt to such variable content. *Handcrafted* solutions perform in correlation with the abilities of an adversary, *i.e.*, highly accurate representations are feasible, but they require much expertise and effort. PhD, instead, presents screenshots of web pages which are by definition an accurate representation of what regular websites' users would see in their browsers.

4.2.2 Back-end Logic

Back-end logic is an umbrella term for code executed on a server that is protected from public access, *i.e.*, it cannot be downloaded or repurposed. Typical use cases that require back-end logic are search results, user profiles, and authorization, *e.g.*, with 2FA. *Template-based* solutions cannot replicate back-end logic. *Handcrafted* solutions are more flexible, yet time-demanding, and existing protective measures of the back-end services cannot always be bypassed, *e.g.*, same origin request policy is in use, or push message services require authorization. PhD, on the contrary, does not require any back-end logic treatment or circumvention, because it interacts with the back-end logic like every web browser does. In other words, unlike most phishing approaches, PhD functions like a "man in the middle" attack.

4.2.3 Browser Plug-in Support

Browser plug-ins serve numerous purposes, *e.g.*, they can provide ad blocking, export videos, keep notes, remember credentials, *etc.* Plug-ins can interact with pages, and therefore manipulate a site's content transparently to the potential victim. This provides new opportunities to circumvent image detection approaches, *e.g.*, by blocking ads, or by injecting on the fly some fake banner overlays. *Template-based* solutions do not support fundamental changes on original pages. *Handcrafted* solutions potentially allow significant changes, but they require numerous additional code modifications which ultimately leads to increased work hours. PhD has full control over the rendered HTML and JavaScript code, and it supports FireFox plug-ins out of the box. For that reason, our system facilitates arbitrary changes on web pages by directly injecting code into the website, by applying browser plug-ins, *e.g.*, ad blockers and dark themes, or by manipulating the screenshot pixel-wise.

4.2.4 Completeness

Completeness refers to the provided functionality compared to the original page. If a website is incomplete people might become suspicious. Relative resource paths are particularly problematic for completeness, since their path only remains valid on the original server, but not on the phishing server. Therefore, either the paths must be adjusted to the resources on the original server, if possible, or all resources must be tracked and copied to the phishing server. *Template-based* solutions try to find and fix problematic resource paths automatically, but they might fail when assembled at run time, or obfuscated in code. *Handcrafted* solutions can take care of such issues, but it is a cumbersome and time-consuming process. PhD does not require any changes because the website's origin is not altered in any way.

4.2.5 Collected Data

Phishing purposefully requires sensitive data to be collected, *e.g.*, user names, passwords, tokens, *etc.* The more data that can be exfiltrated, the better the chances are for an adversary to successfully exploit people. *Template-based* solutions induce constraints on the possible collection of data, *i.e.*, one must strictly use predefined methods for gathering data. *Handcrafted* solutions do not induce any constraints, but time-consuming manual labor is involved. PhD does not require any operations for gathering data as the exfiltration process remains transparent to the victim. All key presses, mouse click coordinates, and potentially all created website screenshots are captured.

4.2.6 Continuous Tracking

Continuous tracking enables observation of users throughout their entire surf sessions. The continuous data stream that web surfers generate is very valuable. For instance, someone could record personal browsing preferences and use them for blackmailing, or collect multiple logins to gather access to different services. Nevertheless, many existing phishing sites hand over control to the real site after the phished data has been gathered. *Template-based* solutions allow multiple phishing pages to be spawned on the same domain, but they cannot serve arbitrary pages on demand. They would require prior setup of every page a user could visit, which is unfeasible. *Handcrafted* solutions suffer from the same limitations. PhD does not require any website preparation, hence it supports the logging of entire surf sessions beyond domain boundaries. For example, if a search engine is set as the landing page, the user can be tracked while entering a search term and choosing an entry from the results, and even while navigating through the desired web pages.

4.2.7 Recency

Recency refers to actuality of phished sites, *i.e.*, whether a phished site reflects arbitrary changes of the original, and how long it takes to reflect them. The more recent a replica is, the more convincing it generally can be for a potential victim. *Template-based* solutions are incapable of automatically integrating changes into phished sites that are already live. For that reason, the process has to be repeated every time the original website is modified. *Handcrafted* solutions could implement an update mechanism, however

this would result in complex code. We assume it is safe to say that they suffer from the same limitations as the template-based solutions. PhD always provides the latest version of a website to the potential victim.

4.2.8 Required Knowledge

Some IT knowledge is required in order to run a phishing campaign. The less knowledge is required, the more people can perform phishing. *Template-based* solutions only require basic web development knowledge, *e.g.*, the ability to install software and to import an existing website into a web framework. The import process can be either performed manually by copy-pasting code into the framework or by importing relevant pages through an assistant that will guide the user through required changes. *Handcrafted* solutions require that an adversary first copies original site content to another server, before site-specific customizations must be applied, *e.g.*, altering HTML forms or JavaScript content. There exists some pre-baked code that performs the extraction of credentials,[2] however various manual adjustments are still required. Hence, thorough web development skills are required. PhD requires only basic IT knowledge, but no web development skills: the only required configuration data are the URL to phish and the URL to be used for the phishing site.

4.2.9 Required Set-up Time

The required set-up time indicates the time it takes to set up a phishing instance. The efficiency increases by providing a shorter set up, *i.e.*, more time remains available for other tasks. *Template-based* solutions require the installation of one or more frameworks, but in general, they try to reduce user interventions wherever possible. They achieve that goal by providing predefined capture schemes that can be applied to original pages, or by providing predefined templates from which a user can choose. Nevertheless, a user must read the documentation to understand all the features required to successfully set up a phishing page. *Handcrafted* solutions consist entirely of manual work, *e.g.*, collecting the resources, adapting the content to the new domain, and injecting phishing code. The required time heavily depends on the user's experience: whereas a novice requires days, an expert might only need a few hours or even minutes. PhD requires only very little time, *i.e.*, a few seconds to start the application.

[2] `https://github.com/mgeeky/PhishingPost`

5

Proof of Concept

While we referred in the previous chapter to the concept called PhD, we discuss in this chapter its implementation and operation. Since the development is still in early stages, we call the implementation a "Proof of Concept" (PoC) prototype.

The system adopts a client/server architecture: the server-side scripted website renderer component is implemented in *Selenium*,[1] and the client-side *JavaScript* code maintains state between a client and the phishing server.

The client tracks the victim's actions, submits them to the server, and renders screenshots with the help of metadata received from the server. In more detail, the victim's keystrokes and mouse click coordinates are tracked and submitted to the server with custom TypeScript code that Aurelia[2] transpiles into JavaScript code. The use of the HTML `canvas` element instead of the `image` tag for displaying screenshots ensures for the client smooth image transitions without any artifacts or other visual glitches, *e.g.*, flickering. The metadata received from the server is processed as follows: the client separates coordinates and embedded text of text boxes, the locations of buttons, and clickable areas of hyperlinks. Next, text boxes and buttons that contain the original text, as well as clickable hyperlinks, are rendered locally on top of the screenshot. The reason for this sophisticated implementation is the authentic look and feel that locally rendered GUI elements provide: they enable user interactions without any delays, provide a consistent user experience

[1] a comprehensive browser automation toolkit, `https://selenium.dev/`

[2] a modern front-end JavaScript framework for building web applications, `https://aurelia.io/`

across other non-phished websites, and they enable support for mobile devices with on-screen keyboards.

The server replays the victim's keystrokes and mouse clicks on the original page by using a scriptable headless web browser, *i.e.*, a web browser with no GUI that provides an API for instrumentation. Due to the superior documentation, we chose the Mozilla Firefox web browser in favor of Google Chrome. After each replayed user action, a new screenshot of the resulting page is sent back to the client.

Several additional features have been implemented on both ends to facilitate a more realistic surfing experience for the victim: i) the title of the original web page is mimicked by using the data within the HTML `<title>` tag, ii) the original website icon, also known as "favicon", is copied from the original location to the server where it is provided to the client, iii) the URL's path is replicated according to the original website's structure, *e.g.*, `https://accounts.google.com/login` results in `http://my-phishing-service.com/login`, iv) pop-ups pointing to resources on the phishing server are supported by using an identifier in each server request which allows the server to differentiate between a client's views, *e.g.*, if a user clicks on a link that spawns a new tab, then the server is able to handle both tabs, and finally, v) optional ad blocking is supported, since ad-loaded web pages can take considerably longer to load and thus would negatively impact the victim's experience.

5.1 Implementation Details

The PoC project consists of two Maven[3] modules: a front-end module that provides all client side features, and a back-end module that offers all server side features. The front-end module runs in the victim's browser and is built as an *Aurelia* application using Aurelia's command-line interface[4] and NPM[5] for the dependency management. To build the front-end module we opted for Gulp.[6] On the other hand, the back-end module is implemented as Java application that handles and processes HTTP-requests originating from the front-end module. We used the *Spring* framework[7] to manage a Selenium instance[8] and web communication. Selenium requires a "web driver" file to control a Firefox headless web browser which can be obtained from Mozilla's website.[9]

In the remaining of this chapter we discuss the client and server side features from a more technical perspective, before we finally discuss the PoC's operation. In cases where the client and the server share feature responsibilities, we discuss them in the subsections that are more relevant to them. The term "user" refers to the person accessing the PoC's generated website, *i.e.*, the victim.

[3] a dependency management system, `https://maven.apache.org`
[4] `https://aurelia.io/docs/cli`
[5] a package manager for JavaScript applications, `https://www.npmjs.com`
[6] a toolkit to automate and enhance a workflow, `https://gulpjs.com`
[7] an application framework, `https://spring.io`
[8] a browser automation toolkit that provides a headless browser component, `https://www.selenium.dev/`
[9] `https://firefox-source-docs.mozilla.org/testing/geckodriver/Support.html`

5.1.1 Client side Features

5.1.1.1 HTML Title and Favicon

The PoC replicates the original website's title and favicon. Whereas the title is accessible through the `document.title` DOM property of the original site, fetching the corresponding favicon is not trivial. The favicon file is either located in the root directory named `favicon.ico` or at the path specified by an HTML `<link>` tag as shown in Listing 1. We locally cache each retrieved favicon as a `png` image and return its link to the front-end. If the original favicon download fails, *e.g.*, indicated by a `HTTP 403: Forbidden` error message, a path to a transparent icon is returned instead.

```
<!-- a favicon specified by the attribute value "icon" -->
<link rel="icon" href="path/to/favicon">

<!-- a favicon specified by the attribute value "shortcut icon" -->
<link rel="shortcut icon" href="path/to/favicon">
```

Listing 1: Two possibilities to specify the favicon location

5.1.1.2 Rendering of Hyperlinks

The PoC re-enables hyperlink interaction on the client, *i.e.*, the hand cursor appears when hovering over a hyperlink, because the transmitted screenshot does not contain any hyperlink information. For clickable elements, the default CSS `cursor` property attribute is `pointer` which we leverage through invisible buttons. We place such buttons at every location that originally contained a clickable element that we could find in the original website's DOM. The corresponding code to find such elements is shown in Listing 2.

```
private List<WebElement>filterClickableElements() {
  List<WebElement> inputElements = driver.findElements(By.tagName("input"));
  inputElements = inputElements
    .stream()
    .filter((WebElement element) -> {
      String type = element.getAttribute("type");
      for (ClickableInputTypes current : ClickableInputTypes.values()) {
        if (current.equalsType(type)) {
          Dimension size = element.getSize();
          return element.isDisplayed() && size.width > 0 && size.height > 0;
        }
      }
      return false;
    })
    .collect(Collectors.toCollection(ArrayList::new));


```

```
18   List<WebElement> elements = driver.findElements(By.tagName("a"))
19     .stream()
20     .filter((WebElement element) -> {
21       Dimension size = element.getSize();
22       return element.isDisplayed() && size.width > 0 && size.height > 0;
23     })
24     .collect(Collectors.toCollection(ArrayList::new));
25   List<WebElement> buttons = driver.findElements(By.tagName("button"))
26     .stream()
27     .filter((WebElement element) -> {
28       Dimension size = element.getSize();
29       return element.isDisplayed() && size.width > 0 && size.height > 0;
30     })
31     .collect(Collectors.toCollection(ArrayList::new));
32   elements.addAll(buttons);
33   elements.addAll(inputElements);
34
35   log.info(String.format("Found [%d] Clickables", inputElements.size()));
36   return elements;
37 }
```

Listing 2: Code to find clickable elements in a website's DOM

5.1.1.3 Rendering of Screenshots

The PoC renders in the user's browser website screenshots received from the server back-end. We could not rely on using the HTML `<img>` tag since that would cause flickering when screenshots are updated. Instead, we had to adapt the rendering process so that the screenshots are rendered in a canvas. The declaration of such a canvas element is shown in Listing 3, and the code responsible for drawing the image onto the canvas can be found in Listing 4.

As Listing 4 reveals, during every screenshot update the method `changeImage` is called where the path for the new image is set. Once the image data is available, `drawImage` renders the image onto the canvas. Because the screenshot should be displayed only once, any previous screenshot is hidden from the view using few lines of JavaScript code. In our experiments, the size of the canvas remained stable as long as the user did not alter the browser window size.

```
1 ...
2 <canvas id="canvas" width="1841" height="951"></canvas>
3 <img src="" alt="" id="image" load.trigger="drawImage()"/>
4 ...
```

Listing 3: Declaration of an HTML canvas image element

```
private changeImage() : void {
  var img = document.getElementById("image") as HTMLImageElement;

  if (img.complete) {
    this.drawImage();
  }
  img.src = this.imagePath;
}

private drawImage(): void {
  var img = document.getElementById("image") as HTMLImageElement;
  var ctx = (document.getElementById("canvas") as HTMLCanvasElement).getContext("2d");
  ctx.canvas.width = img.width;
  ctx.canvas.height = img.height;
  ctx.drawImage(img, 0, 0);
}
```

Listing 4: TypeScript code to draw a screenshot onto a canvas element

5.1.1.4 Rendering of Input boxes

The PoC re-enables input box interaction on the client, *i.e.*, the text selection cursor appears when hovering over an input box or the on-screen keyboard slides in when tapping on an input box displayed by a touchscreen device. This is necessary, because the transmitted screenshot does not contain any input box information.

Similarly to the hyperlink detection, the original website's DOM is traversed to find HTML input elements like `<input type="text">`. We only search for text, password or email HTML input elements, but not for input elements of the type button that disallow all text input. When an input box is found, its coordinates, size, and default values are extracted and transmitted to the front-end which then genuinely displays those elements. The corresponding code to find such elements is shown in Listing 5.

```
private List<WebElement> filterTextInputElements() {
  List<WebElement> elements = driver.findElements(By.tagName("input"));
  elements = elements
  .stream()
  .filter((WebElement element) -> {
    String type = element.getAttribute("type");
    for (TextInputTypes current : TextInputTypes.values()) {
      if (current.equalsType(type)) {
        Dimension size = element.getSize();
        return element.isDisplayed() && size.width > 0 && size.height > 0;
      }
    }
    return false;
  })
  .collect(Collectors.toCollection(ArrayList::new));
  log.info(String.format("Found [%d] input boxes", elements.size()));
  return elements;
}
```

Listing 5: Code to find input boxes in a website's DOM

5.1.1.5 Replication of URL Subpaths

The PoC supports the replication of URL subpaths. For instance, if the PoC runs on the domain `www.phishingserver.com` replicating `www.bank.com`, URL subpaths are replicated as well. That is, `www.phishingserver.com/path?q=query` would be translated on the phishing server to `www.bank.com/path?q=query` and vice versa. For this feature, we forward the current subpath from the headless browser to the user's browser which navigates to it using Aurelia's routing feature.

The relevant Aurelia code is displayed in Listing 6, and the corresponding HTML code in Listing 7. The `configureRouter` method assigns the default HTML template specified in `moduleId` to all future navigation requests. The `route` property is set so that for navigation requests a `path` argument can be provided optionally, *i.e.*, `:` distinguishes the `path` parameter as required token, and `?` makes it optional. The `path` argument contains the complete URL path, *e.g.*, `/personal/profile?query=username`. When the headless browser encounters a new subpath during instrumentation, the back-end sends a route change request to the front-end that contains the desired subpath. Whenever a route change request arrives at the client for the displayed website, the method `originalPagePathChanged` is called which performs the navigation to the desired new path. This navigation enables the client browser's back and forward buttons; a user can navigate back and forth and will still stay inside the same Aurelia application.

```
configureRouter(config: RouterConfiguration, router: Router): void {
  this.router = router;
  config.options.pushState = true;
  config.map([
    { route: ':path?', name: 'default', moduleId: './routes/default', nav: false }
  ]);
}

originalPagePathChanged(newValue: string, oldValue: string) {
  document.documentElement.scrollTop = 0;
  this.router.navigateToRoute("default", path: this.originalPagePath);
  this.needsHtmlUpdate = true;
}
```

Listing 6: TypeScript code for dynamically navigating through URL subpaths

```
<template>
  <button id="wrapperButton" keydown.delegate="updateCommandToKeyDown($event)">
    <div id="wrapperDiv" click.delegate="updateCommandToClick($event)">
      <canvas id="canvas" width="1841" height="951"></canvas>
      <img src="" alt="" id="image" load.trigger="drawImage()"/>
    </div>
  </button>
  <router-view></router-view>
</template>
```

Listing 7: HTML code that uses Aurelia's routing feature

5.1.1.6 User Tracking

The PoC features the tracking of user actions, *e.g.*, mouse clicks and keyboard input, and the replay of such actions in the headless browser on the back-end. We implemented this key feature with two event listeners, one for each the mouse clicks and keyboard strokes as shown in Listing 8. The listeners store the captured information in `command` objects that are forwarded to the back-end using the `update` method.

When the back-end receives the HTTP-request initiated by the front-end's `update` method, it maps the incoming command to a Java command using the attached `type` property as illustrated in Listing 9. In more detail, the `getInstance` method returns the corresponding Java command according to the passed arguments. Supported Java commands are "left mouse click", "keystroke", "back button", "forward button", and "empty". The "empty" command (from the "null object" design pattern) was introduced to avoid potential `null` and `undefined` checks. The prepared Java commands are then forwarded to the `BrowserController`.

Every interaction with the headless web browser is piped through an instance of `BrowserController`, which controls a Selenium web driver instance. After a command is executed by the `BrowserController` a screenshot is taken of the resulting website. The web driver can only take screenshots if the headless browser is not busy, *e.g.*, loading a website. Hence, we had to await the website loading time before we could create a screenshot. The code relevant for that synchronization is presented in Listing 10. The headless browser's current state can be read from the `document.readyState` property.

```
public updateCommandToClick(event: MouseEvent) : void {
  this.command = new MouseClick();
  this.command.xCoord = event.pageX;
  this.command.yCoord = event.pageY;
  this.update();
}

public updateCommandToKeyDown(event: KeyboardEvent) : void {
  if (!(this.command instanceof KeyStroke)) {
    this.command = new KeyStroke();
    this.command.keyList = new Array<string>(event.key);
  } else {
    this.command.addKey(event.key);
  }
  this.update();
  this.command = new EmptyCommand();
}
```

Listing 8: Event listeners for user actions

```
/**
* This enum maps a string received at the REST API to the corresponding Java command
* (class).
*/
public enum CommandTypes {

  EMPTY ("empty", EmptyCommand.class),
  MOUSECLICK ("mouseclick", MouseClick.class),
  KEYSTROKE ("keystroke", KeyStroke.class),
  BACK ("back", NavigateBack.class),
  FORWARD ("forward", NavigateForward.class);

  private final Log log;
  private final String name;
  private final Class equivalentClass;

  CommandTypes(String value, Class equivalentClass) {
      this.log = new Log(CommandTypes.class);
      this.name = value;
      this.equivalentClass = equivalentClass;
  }

  public boolean equalsName(String otherName) {
      return name.equals(otherName);
  }

  public String toString() {
      return this.name;
  }

  public Command getInstance(Object[] args) throws IllegalAccessException,
    ↪ InstantiationException {
      try {
          return CommandFactory.getInstance(equivalentClass, args);
      } catch (CommandSetupException e) {
          log.error("Command setup failed - instead returning an EmptyCommand");
          return new EmptyCommand();
      }
  }
}
```

Listing 9: Available Java back-end commands and their mapping to front-end commands

```
/**
  * Wait up to the moment that the browser has finished loading a webpage, and wait
  * for document.readyState to return "complete".
  * @param driver a WebDriver to control the browser
  */
public void awaitBrowser(WebDriver driver) {
  WebDriverWait wait = new WebDriverWait(driver, 30);
  JavascriptExecutor js = (JavascriptExecutor) driver;
  // wait for browser to react
  Sleeper.sleep(LongProperties.AWAITBROWSER_SLEEP);
  wait.until(driver1 -> js.executeScript("return
    ↪ document.readyState").equals("complete"));
}
```

Listing 10: Web driver synchronization

5.1.2 Server side Features

5.1.2.1 Consistent Image Order

The PoC supports a consistent image order, *i.e.*, concurrent requests from the same user are processed in order. This is mandatory, because HTTP requests show asynchronous behavior. In the example in Listing 11 there is no guarantee that the first asynchronous `sendHttpRequest` method call finishes before the second one. In other words, if an initial request for `image_1` is sent, followed by a request for `image_2`, then the second request might finish before the first one can return a result. Such a race condition would lead to the display of `image_2` instead of `image_1`, and even worse `image_1` would overwrite `image_2` when it arrives although it is older than the already existing image. The code for this ordering measure is listed in Listing 12.

```
this.sendHttpRequest(data).then((result) => {
  // This might not be executed before second "then" block
})
this.sendHttpRequest(data).then((result) => {
  // This might be executed before first "then" block
})
```

Listing 11: Typical example of asynchronous HTTP requests

```
if (this.step <= response.step) {
  this.imagePath = response.html;
}
```

Listing 12: HTTP request ordering

5.1.2.2 Logging

The PoC back-end supports the persistent storage of received user actions and their resulting screenshots. We use *Apache log4j*[10] for logging and we store the generated logs as text and screenshots as `png` files on disk.

5.1.2.3 Oversize Screenshotting

The PoC back-end supports the creation of screenshots that are larger than the currently available screen resolution of the server. If screenshots are created the same size as the user's screen, websites that provide vertical scrolling are cut off in the screenshot. We experimented with two approaches to resolve this problem.

Using AShot. The `ru.yandex.qatools.ashot`[11] package provides a simple interface to generate screenshots from a headless browser instance. It allows screenshots to be taken while scrolling down through an entire website. The working code is available in Listing 13. However, we realized that this approach suffers from two major problems: First, the execution takes multiple seconds, and second, sticky navigation bars on top of a website, *e.g.*, Google's search bar, appear on every screenshot.

```
1 Screenshot screenshot = new AShot()
2   .shootingStrategy(ShootingStrategies.viewportPasting(1000))
3   .takeScreenshot(driver);
```

Listing 13: Screenshot creation with AShot

Using Selenium. Selenium's screenshot provided by the `getScreenshotAs` method is an image of the browser's viewport, *i.e.*, the part of a website that a user sees while surfing the web. Hence, we had to increase Selenium's web browser window size, *i.e.*, we use the reported width from the client and set the height to the webpage's height.

Ultimately, we chose Selenium for screenshotting, because Selenium does not suffer from AShot's limitations.

5.1.2.4 Tab Support

The PoC supports new browsing tabs opened by a user. Websites usually contain links to other webpages and frequently they open in a new tab, *i.e.*, a pop-up. We added functionality to identify and synchronize different tabs between the front and back-end, *i.e.*, we forward each tab identifier from the headless web browser to the front-end so that it can always report on which tab a user action must be executed on the headless web browser. Moreover, we added functionality to store the front-end application's state in cookies before we open another application instance in a new tab on the client. The front-end application's

[10] a Java-based logging utility, `https://logging.apache.org/log4j/2.x/`
[11] `https://mvnrepository.com/artifact/ru.yandex.qatools.ashot/ashot`

state must be stored in cookies, because otherwise the new application instance would not have access to the browsing history, *i.e.*, which website it should display. The back-end code to spawn a new tab on the client using the headless web browser's new tab URL is shown in Listing 14. The rules of URL subpath replication also apply for new tabs: if the new tab's host is different from the application's host the back-end will translate the subpath as described in subsubsection 5.1.1.5.

```
1 if (!ApplicationContext.opensInNewTab(request.getHeader("inNewTab"))) {
2   if (ApplicationContext.canFakeHost(request.getRequestURL().toString())) {
3     getController().openURL(host, path);
4   } else {
5     getController().openPath(path);
6   }
7 }
```

Listing 14: Code responsible for the new tab navigation

5.1.2.5 Firefox Plug-in Support

The PoC headless browser in the back-end supports Firefox plug-ins that can interact with the requested websites. To increase the flexibility of our solution, we decided to implement plug-in support for the headless browser. In our setup we used an ad blocker plug-in that removes ads on the fly from requested websites and at the same time it increases the rendering speed, because many ads are well-known for increasing the loading times of websites.

This feature can be used by copying an `xpi` extension file into the PoC's plug-in directory and by registering the plug-in in the used Firefox profile. The corresponding code is shown in Listing 15.

```
1 FirefoxOptions firefoxOptions = new FirefoxOptions();
2 FirefoxProfile firefoxProfile = new FirefoxProfile();
3 if (ApplicationContext.config.getBoolean(BooleanProperties.ADBLOCK)){
4     firefoxProfile.addExtension(new File(ApplicationContext.config.getString(
5     ↪ StringProperties.ADBLOCK_PATH)));
6 }
7 firefoxOptions.setProfile(firefoxProfile);
8 driver = new FirefoxDriver(firefoxOptions);
```

Listing 15: Code to load a Firefox plug-in into a web driver instance

5.2 Operation

Only a few prerequisites are mandatory for the PoC: i) a server that contains a multi-core processor and must be accessible from the internet through a fast network connection, preferably faster than 20 MBit

per second per user, ii) a valid domain name must refer to the server, iii) a *Java 8* or higher runtime and a recent version of *Firefox* must be installed, iv) the platform dependent *geckodriver* file[12] and the executable PoC `jar` file must be stored on disk, and finally, v) the same applies for the relevant `xpi` *Firefox* plug-in file, if the optional ad blocking is used.

The PoC server supports the use of different configuration profiles that allow fast and easy transitions between different setups. Each profile is stored in a property file in the folder `/BOOT-INF/classes/config` in the compressed archive `PhishingOnDemand.jar`. The default configuration will be used, if no profile is specified during startup.

Four properties must be specified to operate the phishing system: i) `backendBase` and `aureliaBase`, which must both contain the phisher's domain URL, *e.g.*, `http://www.my-phishing-service.com/`, ii) `entryPoint` which specifies the site to spoof, *e.g.*, `https://www.e-banking.com/`, and finally, iii) `webdriver.gecko.driver` which specifies the path to the *geckodriver* file, *e.g.*, `installs /geckodriver`.

The following seven properties are optional: i) `adblockPath` which specifies the path to the *Firefox* ad block library file, *e.g.*, `installs/plug-in.xpi`, ii) `adblock` which toggles the ad blocking, iii) `debug` which toggles additional debug console output, iv) `headless` which toggles the visibility of the browser used for creating the screenshots, v) `mouseClickSleep` which specifies the time to wait in milliseconds before a screenshot is taken of the final page reached after a user performed a mouse click, vi) `keyStrokeSleep` which specifies the time to wait in milliseconds before a screenshot is taken of the final page reached after a user performed a keystroke, and finally, vii) `awaitBrowserSleep` which specifies the time to wait in milliseconds before the *geckodriver* is allowed to access the newly instantiated browser instance.

After fulfilling all prerequisites and adapting the configuration file, the application can be started by executing the command `java -jar PhishingOnDemand.jar`.

[12] `https://github.com/mozilla/geckodriver/releases`

6

Empirical Study

We disguised the empirical phishing study as a website usability study to not raise any suspicions. The seven participants with diverse backgrounds in different age groups, *i.e.*, from computer science students to retired business owners, have not been introduced to PhD. The goal of this study is to assess the potential threat that arises from PhD by evaluating its weaknesses when in active use, *i.e.*, the differences to original websites noticed by the participants.

6.1 Questionnaire

The questionnaire we used can be found in the PoC's public Github repository and comprises three sections. In the first section, we gathered personal information, *e.g.*, age, level of education, profession, or the experience using the web. In the second section, we let the participants interact with different websites and after each interaction session we questioned them about their experience, the usability, and other observations they drew. In the last section, we gathered more information regarding the familiarity with the previously visited sites, and at last, to not raise any attention during the study, their knowledge of phishing. If they lacked some experience with phishing, we explained what it is, and finally, we questioned them in retrospect for each task the perceived likelihood the relevant site was phished.

6.2 Performing the tasks

The participants had to solve five tasks, where each task focuses on one particular kind of website, *i.e.*, originals without any modifications, originals replicated by PhD, or traditional phishing websites.[1] Hence, the tasks differ in the use of tools, *i.e.*, one task did not involve any tool, one involved a traditional phishing tool, and three involved our PhD system. In terms of complexity, one task was only about visual inspection, one about navigation through websites, and three about entering credentials into login pages. Except for the first introductory visual inspection task, the order of the tasks has been randomized to mitigate any potential bias in the results. We set a time limit for each task so that we expect the participant to be able to solve the task, but unable to inspect every single detail of the website. In order to let the participants mainly focus on the website itself, all tasks were performed in full screen mode so that the address bar is invisible. While the supervisor was preparing a task, the participants were not allowed to watch the computer screen. All credentials the participants were supposed to use have been prepared by the authors to avoid any unintended data leaks.

The individual tasks were as follows. 1) Visual inspection of a static website with PhD: each participant should look at Google Search for a maximum of ten seconds, 2) Static website with PhD: on Google Search, each participant had to search for "Swiss Radio and Television (SRF)", to navigate to that page, and if there was some time left, they were encouraged to continue exploring that website for in total up to 30 seconds, 3) Login website with PhD: each participant was tasked to search for "Amazon Germany", to navigate to that page, and finally, to log in with the given credentials for up to 90 seconds, 4) Login site without any modifications: each participant was told to search for "Facebook", to navigate to that page, and finally, to log in with the given credentials for up to 90 seconds, 5) Login site with a traditional phishing tool: each participant had to log in through the presented Google GMail login website by using the provided credentials for a total of 60 seconds.

Immediately after completion of each task, we asked every participant the following five questions: i) Do you have any experience with the visited website, and if so, what kind of experience? ii) What was the website's level of usability on a scale from one to five, where one refers to very low and five refers to very high? iii) Was there something you really liked concerning the usability? iv) Was there something you really disliked concerning the usability? v) Is there something that could be considered to improve usability?

6.3 Findings

In the findings, we discuss the feedback from the participants where we specifically focus on their perception of phished websites. The received feedback matches the three categories usability, authenticity, and further observations.

[1] `https://github.com/ashanahw/Gmail_Phishing`

6.3.1 Usability

We asked all participants after every task to rate the perceived usability level on a Likert-scale between one (very low) and five (very high). The participants rated the usability on average with a score of four (*i.e.*, high) for the original web page as well as for the web page from a traditional phishing tool. When we average the usability scores of the other three tasks that used PhD, we see a score of 3.6 (*i.e.*, medium to high).

In the traditional phishing website, some embedded hyperlinks and the focus order are broken. Moreover, after stealing the credentials, the website tries to redirect the user to the original website, which requires the participants to log in a second time. One participant noticed the focus order bug, and another one noticed that this website was quicker than PhD. Surprisingly, only one participant was irritated that the credentials had to be entered twice and reported that as "phishy". However, those problems did not affect the perceived usability.

Based on the participants' responses for PhD, the lower usability score has two main reasons: First, sometimes our tool produced unexpected results, and second, our tool introduced a page load delay depending on the complexity of the requested website. Although our tool works with most of the existing websites, it still suffers from occasional glitches. For instance, issues could lead to corrupted text input when a participant is typing very fast, because in the current implementation the rendering of text boxes is not yet synchronized with the screenshots. This confused some participants and forced us to manually reload the affected website. Moreover, the positioning of text boxes could be out of sync with the transferred image. One participant noticed that problem after completing a *Google Search* task. The additional delay before a website is displayed has also been reported: On the one hand, two participants criticized the longer than usual loading time once they clicked on a hyperlink to another website. On the other hand, one participant preferred that the results of his *Google Search* were displayed all at once, *i.e.*, no changes to the page layout occurred during the rendering process. This is a result of the simplified rendering on the client that only needs to show one static image instead of thousands of vectorized characters, and HTML elements manipulated by JavaScript code at run time.

We conclude that the loading time of a web page has a significant impact on how people rate a page's usability, but differences in usability levels do not make people aware of any malicious activity, except when people have to enter their credentials more than once.

6.3.2 Authenticity

After the experiment the participants were asked to judge the likelihood of the tested websites being phished replicas for the previously seen websites, again by using a Likert-scale from one (very unlikely to be a phishing site) to five (very likely to be a phishing site). We received for our PhD sites on average a score of 2.1, whereas the original Facebook page received 2.4 and the traditional phishing page achieved 2.7.

Our screenshots were the most authentic to participants and replicas barely distinguishable from the

originals. To our surprise, the original Facebook was rated more likely to be phished. That we can only explain by the many phishing scams that became public in the news which raised attention to this specific website. As expected, the traditional *Google Gmail* phishing site used in our experiment received the highest Likert-score.

Many of the participants were rather experienced using Google and Facebook, *i.e.*, six participants use Google multiple times a day, whereas one participant uses Facebook daily and two at least once a week. The design of the traditional *Google Gmail* phishing site we used in our experiment dates back to March 2017, therefore the style clearly differs from recent iterations. No participant, not even the one who uses this service on a daily basis noticed that, even when entering the credentials for the second time on a page with a different design. On the contrary, one participant considered a task spoofed, because URLs on Google Search's result page were not green like on his own computer. However, this assumption is incorrect, because Google recently updated their design.

In conclusion, it appears that the participants are not paying much attention to the design of a website. It seems that they associate the term "phishing" rather with particular websites of large companies, than with sites of small to medium sized businesses. Hence, they more likely report phishing activity on those websites, even when no attacks are performed. All participants genuinely do not know how to decide if a site is phished or not.

6.3.3 Further Observations

One participant felt responsible for not completing a task even though the application did not correctly render the screenshots. Another participant mentioned that the traditional phishing site started to look phishy when "unexpected messages" appeared in the browser, *e.g.*, a password save dialog box should not appear on regularly visited sites. Finally, one participant did not expect any phishing on a PhD site, because there existed no login on the viewed page.

7

Limitations, Mitigations and Opportunities

In this section we discuss the tool's limitations, potential mitigation strategies, and opportunities beyond phishing. Since we aimed during development for feature support rather than efficiency, our prototype still provides several opportunities for optimizations.

7.1 Limitations

The tool currently suffers from inherent limitations such as delays or limited interactivity on dynamic content for websites that heavily use scripting. Nevertheless, these problems can be mitigated with additional engineering effort.

7.1.1 Computational Delays

For our implementation we currently use a sophisticated architecture that enables rapid development of features, but requires a comprehensive software stack including several different high-level frameworks, *e.g.*, Aurelia, Selenium, Spring Boot, and Tomcat. Hence, an improvement would include hardware acceleration support, *e.g.*, for image compression, and more use of low-level programming languages causing less computational overhead. Next, on the proxy server, every website request must be at least partially finished before a screenshot can be sent back to the client, whereas a regular client can start displaying a website right away as soon as data is received. For this problem, a solution could be to use

video streams instead of screenshots similar to the implementation used in game streaming services. By using such technology, the user would see every step of the website rendering which is very similar to the regular browsing experience. Moreover, the server needs to fully execute every incoming website request. Because complex websites require much processing and RAM for display, *e.g.*, up to several hundred megabytes for a single page, the hardware must be extraordinary powerful. We recommend for up to five concurrent users at least a recent CPU with four cores, 16 GB of memory, a fast SSD, and a dedicated graphics card that supports hardware acceleration to achieve a sustained performance. If more concurrent users are desired, the PhD service could be moved to a cloud service provider where additional instances can be launched on demand to dynamically suit the needs.

7.1.2 Domain Use

The evaluation, reasonable selection, and registration of domain names is essential for successful phishing campaigns. However, PhD does not support domain-related tasks to make the tool impractical for adversaries.

7.1.3 Transmission Delays

PhD represents a man-in-the-middle system which acts as an image-based HTTP proxy to the internet. Inherent to this architecture, the required network traffic increases in most cases substantially, *i.e.*, screenshots of websites usually are in the megabytes of data compared to text and code that remain rather in the kilobytes. Generally, this increase in data that needs to be transmitted to the client can increase the time required to display a website. This problem can be reduced by using high-bandwidth network interconnects for the server and, if possible, for the clients. Another approach would be to use a better image compression algorithm than *portable network graphics (png)* which could massively reduce the amount of transmitted data. However, this change might also reduce the visual quality of the screenshots since `png` is a lossless algorithm. Finally, we found that on entry-level devices with very low CPU performance, *e.g.*, using an Intel Atom Z3735F CPU, PhD can even reduce the time required to display complex websites, because the native rendering of complex websites consumes substantially more CPU time than only rendering an image where a powerful server handles the workload.

7.1.4 Limited Interactivity

Screenshots are only updated on the client side after user actions, *i.e.*, mouse clicks or keyboard input. As a result, dynamic elements such as CSS animations and videos will be transformed into static images and therefore no longer be dynamic. To cope with this problem, we support parameterization of different time-out values through the configuration file, *e.g.*, the time to wait before creating a screenshot when loading a website, or the time to wait when a user provides continuous input. Again, we could use video streaming that would deliver all dynamic website animations in almost real time to the client.

7.1.5 Phishing Campaigns

Professional phishing tools allow one to create and send phishing emails to individuals or address groups. They provide mail templates and provide placeholders for personal formulations, *e.g.*, the victim's name, which are then automatically replaced before sending the message. In that regard, functionality to import large data sets from Comma-Separated Value (CSV) files is present. For instance, such CSV files can contain victims' email and postal addresses, phone numbers, and countries of residence. Moreover, they collect statistics, *e.g.*, the time a phishing email has been dropped, when the email has been opened by the victim, or if and when the contained hyperlink has been accessed. To check if an email has been opened, the tools usually embed a small invisible unique image, *i.e.*, tracking pixel, that is requested from the phishing server at the time the email is presented to the victim. PhD does not support phishing campaigns to make the tool impractical for adversaries. Hence, we neither support phishing emails, nor do we leverage any statistics about a campaign's success rate.

7.2 Mitigation Strategies

Anti-phishing measures aiming at the detection of text, source code, or website images have no impact on PhD, because these file-based resources are not used in their original form. As a result, we see two options to mitigate this phishing threat: run time flow analysis of user input and domain analysis.

7.2.1 Flow Analysis

Our approach forwards all user input to the network socket. This behavior could be leveraged to decide whether the website has a malicious intent. Since the browser has full control over user input, JavaScript execution, and network communication, it could label data entered by a user and track the labels across the *Document Object Model (DOM)* and even through the JavaScript execution engine.[1] If the browser detects that a website continuously sends labelled data over the network, it should display a warning message to the user. Revoking the access to said site might be inappropriate, because some benign websites can reveal such behavior, *e.g.*, multiplayer games.

7.2.2 Domain Analysis

Our implementation provides exact visual replicas of websites, but their domain names might still differ. This situation can be leveraged by a detection tool. Suppose that a user already possesses credentials to be vulnerable to a phishing attempt. Consequently, the user already had to visit certain benign websites in order to create a user account on them. During that process, the browser could start to "know" how the original websites look like, *e.g.*, by building a ground-truth dataset based on screenshots. If later a website screenshot looks similar to one from the ground-truth dataset but uses a different domain name, the browser should block access to the site and show a warning message.

[1] a browser extension that provides taint analysis of string values in JavaScript, `https://github.com/ollseg/ttt-ext`

7.3 Opportunities

PhD can also be used for improving security: browsers do not have to execute insecure content locally, and since it is harder to filter image-based traffic, it could be used as a proxy server in places with restricted internet access.

7.3.1 Improving Browsing Security

Due to the architecture, all potentially insecure content is executed server side. Hence, if a website contains exploits or telemetry tools, those elements would cause less harm, because the exploits are executed remotely and the telemetry data would become corrupt as different users share the same browser configuration. Regarding exploits, for example, if the remote browser crashes the user only sees an empty page or a time-out notification depending on the severity of the attack. The user can afterwards restart the server to reestablish the connection, or if preferred, reset the environment back to defaults to remove any potential traces of the attack.

7.3.2 Bypassing Filtering of HTTP Content

In some countries web requests are continuously monitored for black-listed domains or keywords, and the use of VPNs is prohibited. This framework can resolve this issue when used with unsuspicious domains that provide the PhD service which transmits images instead of textual content. Such content is much harder to categorize and block in an automated manner. Image detection algorithms would need to be deployed, and one could easily change the image protocol to cope with any changes, similar to the Captcha protection system.[2]

[2] `https://www.google.com/recaptcha/intro/v3.html`

8

Threats to Validity

A major threat to validity is the selection of the participants in this study, *i.e.*, whether our selection of participants is representative of the public. We strived to include people with various backgrounds, ages, and internet knowledge. Moreover, we gathered demographic information to see if we can find any correlations. As with any Likert-scale, we cannot guarantee the accuracy and comparability of the participants' responses. On principle, these responses are subjective and can be influenced by their feeling at that time. In addition, the experiment might have induced stress to some people which further impacts the outcome. Since our implementation is not bug free, some bugs did occur during a few experiments. Although we immediately took countermeasures, this could still influence their response. We did not perform the experiment with every web page in the internet, but only five. However, we tried to select websites with high impact to receive reasonable results. We might have misinterpreted or misunderstood responses of the participants. We tried to mitigate that problem by reviewing our notes after the experiment with every participant. We did not yet implement and validate our proposed optimizations, hence we can only hypothesize about their benefits and compare the expected effects with other well-researched implementations found in popular software. There is a threat to construct validity through potential bias in our expectancy.

9 Conclusion

Phishing is growing in sophistication, but is still rather primitive and technically limited. Specifically the phishing websites are static and not dynamically updated. We identified a new potential threat that we call PhD, where a website is dynamically mimicked for phishing purposes. We show with a proof of concept prototype and a user study that this threat is real. Moreover, we explain inherent limitations, and identify some possible ways to mitigate this new threat. Finally, we show two benefits our tool can offer when used in a benign setup, *i.e.*, the improvement of browsing security and the bypassing of HTTP content filters. We believe that this threat requires more attention, especially when considering the emerging ultra-broadband network technologies, *i.e.*, fiber fixed-line and 5G cellular networks. Hence, we encourage the security community to focus more on the visual aspects of phishing websites, *i.e.*, developing more robust visual similarity algorithms specifically for websites that are able to detect the changes applied to an original site, *e.g.*, in the layout, text, or graphics.

Bibliography

[1] D. Birk, S. Gajek, F. Grobert, and A.-R. Sadeghi. Phishing phishers-observing and tracing organized cybercrime. In *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*, pages 3–3. IEEE, 2007.

[2] D. Canali, D. Balzarotti, and A. Francillon. The role of web hosting providers in detecting compromised websites. In *Proceedings of the 22nd international conference on World Wide Web*, pages 177–188, 2013.

[3] K. L. Chiew, E. H. Chang, W. K. Tiong, et al. Utilisation of website logo for phishing detection. *Computers & Security*, 54:16–26, 2015.

[4] M. Cova, C. Kruegel, and G. Vigna. There is no free phish: An analysis of "free" and live phishing kits. *WOOT*, 8:1–8, 2008.

[5] Q. Cui, G.-V. Jourdan, G. V. Bochmann, I.-V. Onut, and J. Flood. Phishing attacks modifications and evolutions. In *European Symposium on Research in Computer Security*, pages 243–262. Springer, 2018.

[6] X. Han, N. Kheir, and D. Balzarotti. The role of cloud services in malicious software: Trends and insights. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 187–204. Springer, 2015.

[7] X. Han, N. Kheir, and D. Balzarotti. PhishEye: Live monitoring of sandboxed phishing kits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1402–1413, 2016.

[8] M. Hara, A. Yamada, and Y. Miyake. Visual similarity-based phishing detection without victim site information. In *2009 IEEE Symposium on Computational Intelligence in Cyber Security*, pages 30–36. IEEE, 2009.

[9] J. Hong. The state of phishing attacks. *Communications of the ACM*, 55(1):74–81, 2012.

[10] J. Hong, T. Kim, J. Liu, N. Park, and S.-W. Kim. Phishing URL detection with lexical features and blacklisted domains. In *Adaptive Autonomous Secure Cyber Systems*, pages 253–267. Springer, 2020.

[11] A. K. Jain and B. Gupta. PHISH-SAFE: URL features-based phishing detection system using machine learning. In *Cyber Security*, pages 467–474. Springer, 2018.

[12] S. KP, M. Alazab, et al. Malicious URL detection using deep learning. 2020.

[13] W. Liu, X. Deng, G. Huang, and A. Y. Fu. An antiphishing strategy based on visual similarity assessment. *IEEE Internet Computing*, 10(2):58–65, 2006.

[14] H. McCalley, B. Wardman, and G. Warner. Analysis of back-doored phishing kits. In *IFIP International Conference on Digital Forensics*, pages 155–168. Springer, 2011.

[15] D. K. McGrath and M. Gupta. Behind phishing: An examination of phisher modi operandi. *LEET*, 8:4, 2008.

[16] T. Moore and R. Clayton. Examining the impact of website take-down on phishing. In *Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit*, pages 1–13. ACM, 2007.

[17] T. Moore and R. Clayton. Evil searching: Compromise and recompromise of internet hosts for phishing. In *International Conference on Financial Cryptography and Data Security*, pages 256–272. Springer, 2009.

[18] A. Oest, Y. Safei, A. Doupé, G.-J. Ahn, B. Wardman, and G. Warner. Inside a phisher's mind: Understanding the anti-phishing ecosystem through phishing kit analysis. In *2018 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–12. IEEE, 2018.

[19] ProofPoint. State of the phish, annual report, 2020.

[20] R. S. Rao, T. Vaishnavi, and A. R. Pais. CatchPhish: detection of phishing websites by inspecting URLs. *Journal of Ambient Intelligence and Humanized Computing*, 11(2):813–825, 2020.

[21] A. Zamir, H. U. Khan, T. Iqbal, N. Yousaf, F. Aslam, A. Anjum, and M. Hamdani. Phishing web site detection using diverse machine learning algorithms. *The Electronic Library*, 2020.

[22] Y. Zhang, J. I. Hong, and L. F. Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th international conference on World Wide Web*, pages 639–648, 2007.

A

Anleitung zum wissenschaftlichen Arbeiten

The Anleitung consists of the conference paper “Phishing on Demand”.[1]

P. Gadient, P. Gerig, M. Ghafari, and O. Nierstrasz. Phishing on demand.

Planned for submission to *The 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020.

[1] `http://scg.unibe.ch/download/supplements/Phishing-on-Demand-(working-draft).pdf`