Trapped in Transparency: Analyzing the Effectiveness of Security Defenses in Real-World Scenarios

by

Purv Rakeshkumar Chauhan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2022 by the
Graduate Supervisory Committee:

Adam Doupé, Chair
Youzhi Bao
Ruoyu Wang

ARIZONA STATE UNIVERSITY

May 2022

ABSTRACT

Honeypots – cyber deception technique used to lure attackers into a trap. They contain fake confidential information to make an attacker believe that their attack has been successful. One of the prerequisites for a honeypot to be effective is that it needs to be undetectable. Deploying sniffing and event logging tools alongside the honeypot also helps understand the mindset of the attacker after successful attacks. Is there any data that backs up the claim that honeypots are effective in real life scenarios? The answer is no.

Game-theoretic models have been helpful to approximate attacker and defender actions in cyber security. However, in the past these models have relied on expert-created data. The goal of this research project is to determine the effectiveness of honeypots using real-world data. So, how to deploy effective honeypots? This is where honey-patches come into play. Honey-patches are software patches designed to hinder the attacker's ability to determine whether an attack has been successful or not. When an attacker launches a successful attack on a software, the honey-patch transparently redirects the attacker into a honeypot. The honeypot contains fake information which makes the attacker believe they were successful while in reality they were not.

After conducting a series of experiments and analyzing the results, there is a clear indication that honey-patches are not the perfect application security solution having both pros and cons.

DEDICATION

My heartfelt thank you to Adam, for giving me the opportunity to work with him, without whom this project/thesis would not exist. It all started when I took his undergraduate cybersecurity course, which made me interested in the field of cybersecurity. Then gave me an opportunity to be his undergraduate teaching assistant, finally letting me work at the SEFCOM lab as a research assistant to further pursue my passion.

Thank you to Fish for always being available to help or answer any questions even at the middle of the night.

Thank you to Tiffany and Yan for supporting me during my entire journey at the lab.

Along the journey, I faced many hardships, but all the professors at SEFCOM made sure those hardships did not bring me down. Moreover, opened doors to opportunities which I could not have imagined otherwise.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Computer systems and networks are vulnerable to a wide range of cyber threats from malicious actors and various attack vectors. Data needed to detect these threats is spread across multiple systems and devices. Malicious agents use different techniques to carry out these kinds of attacks and also to conceal their actions. Current methods are not able to detect these attacks in a reliable manner. Relevant data which is valuable to detect these kinds of attacks consistently outnumbers total storage, bandwidth and processing capability greatly. Due to such a huge difference, cyber defenders are looking for tools to strategically allocate resources to target data which is actually valuable. Furthermore, current tools are unable to detect innovative techniques and vectors in a proactive manner. Here, the challenge is to analyse data that is spread across different systems and networked devices in an effective manner.

In the past, game-theoretic models have been proposed to generate attacker and defender scenarios. Yet these models have relied upon "expert-created" data [17] or made assumptions [5, 13] about the data. However, this work aims at gathering data rather than creating it, specifically through Capture The Flag (CTF) environments. CTF's are exercises performed individually or in groups to learn or test a variety of cyber security skills. The CTF environments in this work are created to simulate security implications in real life wherein the participants act as attackers and intentionally vulnerable programs work as defenders.

The goal is to be able to create prototype components to run CTF style experiments and integrate them into an existing framework known as iCTF [31, 32] developed by researchers at University of California - Santa Barbara (UCSB) and

Arizona State University (ASU). Then, this would allow experiments to be run with a sizable number of people to gather enough data. This data can be used to create game-theoretic models which may in turn be used to create secure systems. These systems can leverage those data-driven models to go beyond traditional security methods and venture into the realm of building proactive defense systems that can foresee attacks and respond in a game-theoretic fashion, thereby establishing data-driven decision-making and decision-driven data gathering as the backbone of proactive defense mechanisms.

After successfully conducting several experiments over a period of 11 months, meanwhile also making changes to the infrastructure, honey-patches outperfomed the other security defense by a huge margin. But this does not give a solid conclusion that honey-patches are the perfect solution to solve cyber security issues. A solid conclusion can only be derived after providing this gathered data to formulate a game-theoretic model to find the most effective defense mitigation. Although, an intuitive discussion and analysis is done to come up with a preliminary conclusion considering the upsides and downsides of honey-patching. Despite having several advantages of honey-patching as a deception technique for defenders, it has more downside which outweighs the advantages like being detectable if carefully crafted payloads are used for exploitation. Additionally, they require more resource overhead which make the honey-patched process slower, setting off an alarm in the attacker's mind.

Chapter 2

BACKGROUND

Many technologies and frameworks developed by various organizations have been used in this work. Some of them are discussed below:

2.1 ICTF-Framework

Simulating real-life scenarios is a core ideology for this work. One way to do this is by using Capture The Flag (CTF) environments. The framework that has been used for simulating these scenarios is known as the *ictf-framework*. The *ictf-framework* is primarily used to host the iCTF competition every year by team Shellphish.

The iCTF is an attack-and-defense style CTF wherein participants are each given access to a server on a network that is running a set of vulnerable programs that shall henceforth be referred to as services. The participants then launch attacks against each others servers to exploit the vulnerabilities they have found. Also, their duty is to properly patch their own services so that they are protected against exploits coming from the other teams and while still functioning normally.

2.2 Application Security Vulnerabilities

Selecting vulnerabilities is the next step. Since the goal is to simulate real-life scenarios, vulnerabilities were to chosen which were still very prominent in current software applications. These vulnerabilities should also be exploitable in a decent amount of time and be able to wreck havoc on the victim machines, since that is what usually happens during large scale cyber-attacks.

2.2.1 Command Injection

Command injection is an attack wherein an attacker can send arbitrary commands which get executed on a host operating system. They are possible due to improper input validation in a vulnerable application.

2.2.2 Buffer Overflow

Buffer overflow vulnerability can occur when a program attempts to put more data in a buffer than it can hold, this data can overwrite memory which could result in corrupt data, crash the program or cause the execution of malicious code.

2.3 Application Security Mitigations

Defense mitigations need to be chosen to test their effectiveness. Various cyber defense technologies are available and are commonly used, those include Antiviruses, Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), Firewalls and many more. Snort is chosen to compete with Honey-patching, due to the reasons discussed below.

2.3.1 Snort

Snort is a popular network intrusion detection and prevention system. It uses rule-based filters to detect different types of attacks like buffer overflows, Denial of Service (DoS) attacks, OS fingerprinting prevention and many more.

It works as a packet sniffer, which allows monitoring of all incoming packets and identification of those that could be potentially dangerous. It uses rules which are quite easy to create and implement. It is also compatible with most operating systems and network environments.

The reason to choose Snort is it is highly configurable and easy to deploy on Linux distributions (iCTF-Framework runs on Linux distributions)..

2.3.2 Insider

Currently, software security updates result in fixing security vulnerabilities and bugs. This puts attackers at an advantage because now they know that the vulnerabilities in the software have been fixed. This information results in attackers dedicating time to finding new vulnerabilities and subsequently using them for successful attacks, which ultimately boosts their confidence in stolen secrets and potential sabotage as a result of their attacks[3].

A unique approach has been developed by security researchers at IBM to address this problem. It is a methodology for reformulating a large category of security patches into honey-patches that provide equal security, but make it extremely difficult for attackers to tell whether their attacks were successful or not[3]. When an attempt to exploit a known vulnerability is detected, the honey-patch effectively and transparently sends the attacker to a sandbox environment, allowing the attack to succeed. This leads the attacker to believe their attempt was a sucess. The sandbox environment could contain software monitors to capture critical attack information and bogus files that mislead attackers.

A patch management model is implemented to allow for swift injection of software patches into live programs without disrupting production workflows, as well as transparent sandboxing of suspicious processes for counterintelligence and threat information collecting. This solution is practical and easy to deploy. It is implemented as *just-in-time* (JIT) patching for efficient security fixes[4].

The JIT implementation for honey-patching is called *insider*[4]. Figure 2.1 shows the process of injecting a patch into a running application (step 1-3) and the response

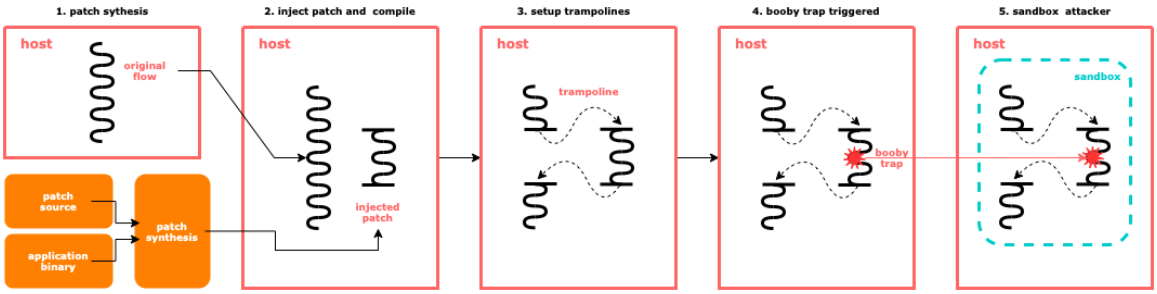

Figure 2.1: System Design For Working Of Insider [4].

triggered by that patch (steps 4-5). In step 1, a patched function (or functions) is developed to replace a vulnerable application's original function (or functions). The patch is synthesized into bitcode, and symbols are retrieved from a copy of the application binary. The patch is injected into the target program's memory space after it has been synthesized (step 2). The patch is subsequently compiled into native code on a separate execution thread inside the process, and linked against the application's global symbols using the symbol mappings from step 1. The execution of the program is momentarily paused to introduce a trampoline from the vulnerable function, which is then replaced with the patch as shown in step 3. When an attack payload is detected in the patched function (step 4), it triggers a user-defined action, in this case, sandboxing in a *chroot* jail (step 5) [4].

2.4 Data Collection Tools

One of very important tasks is to choose good quality data collection tools, which would be used to collect valuable information/data on the attacker and defender machines.

2.4.1 Tcpdump

tcpdump is a network packet analyzer tool. It can capture network traffic by reading packets flowing through network cards. It also has the capability to write

these captured packets into standard output or into files for further analysis.

tcpdump is chosen because attacker and defender virtual machines are connected on a network. It has the ability to capture and store network traffic flowing between separate machines on a network, and this network traffic can then be used for network packet analysis post experiment.

2.4.2 SysFlow

According to *IBM* researchers who developed *SysFlow*, it is "a new system telemetry format and tool suite for monitoring system behavior for scalable security, compliance, and performance analytics. *SysFlow* encodes the representation of system activities into a compact format that records how applications interact with their environment. It connects process behaviors to network and file access activities, providing a richer context for analysis" [28]. *SysFlow* is deployed on *ictf-framework* to capture attack data during the experiments.

SysFlow is chosen because it also has the ability to capture both host and container based application workflows. This provides a rich context for post exploitation analysis.

Chapter 3

INFRASTRUCTURE

The majority of the infrastructure has been developed and integrated into the *ictf-framework*. The infrastructure developed for this work include vulnerable services to be used during experiments, also how defense mechanisms are setup and deployed, which is discussed below:

3.1 Adapting ICTF-Framework

Several modifications have been made to the current implementation of the ictf-framework, most of which include the removal of unused resources and the addition of implementation for setting up defenses and gathering data which are discussed later.The goal of a participant is to read a pre-generated string known as the *flag*. In this work, it is assumed that there is only one attacker, and the vulnerable services are deployed on different machines with varying defenses (Figure 3.1). There is only one *flag* which is placed in the root (/) directory of the host operating system in */flag*. It can only be read by the root user, thus the vulnerable services also have the SUID bit set to 1.

Figure 3.1 shows the implementation used for conducting the experiments. The router acts as a gateway between all the *teamvms*. The participant who acts as an attacker is given access to the *teamvm1* machine via SSH. They can interact with the vulnerable services only via the *teamvm1* machine. All the *teamvms* run a similar environment (Ubuntu 18.04) along with having copies of docker images for all the vulnerable services. Hence, the attacker has access to copies of all the services being used in the experiment to analyze and develop exploits for the services.

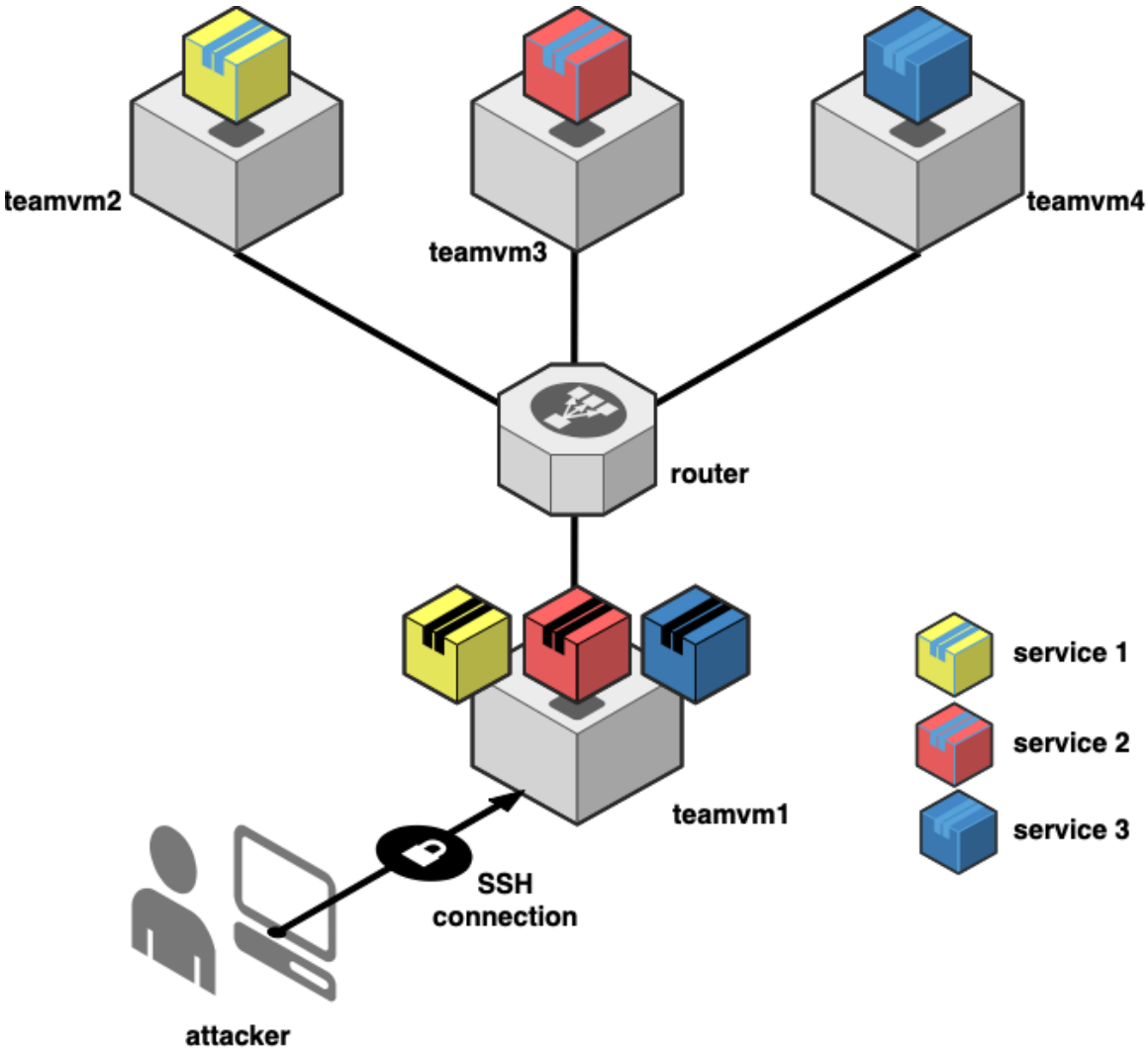

Figure 3.1: Infrastructure Setup For Experiments.

3.2 Vulnerable Services

To be able to analyze the effectiveness of different defense mechanisms with greater accuracy, selecting vulnerabilities to be tested in the experiments is critical. The vulnerabilities being tested are specifically command injection and buffer overflow, which are still prominent in modern security.

All the services discussed have been implemented in C and dockerized to isolate the services from other services and the host. Figure 3.2 is a snippet of the makefile

```
1  CFLAGS += -Wall -O0 -fno-omit-frame-pointer -Wno-deprecated-
     declarations -D_FORTIFY_SOURCE=0 -no-pie -Wno-format -Wno-format-
     security -z norelro -z execstack -fno-stack-protector
2  all: service
3
```

Figure 3.2: Code Snippet Of The Makefile Used For Compiling.

```
1  Arch:     amd64-64-little
2  RELRO:    No RELRO
3  Stack:    No canary found
4  NX:       NX disabled
5  PIE:      No PIE (0x400000)
6  RWX:      Has RWX segments
7
```

Figure 3.3: Output Of Running *checksec* On A Compiled Service Binary.

used for compiling the services.

Figure 3.3 shows the output of *checksec* when ran on a compiled binary.

The 64-bit binary is compiled with all the modern security mitigations turned off. The reason to do this is that it provides a better insight into how effective the defense mechanisms being tested are.

3.2.1 Backup

backup is the service deployed on *teamvm2*, which allows users to store and retrieve data that is stored as files on the host system. The intended vulnerability is placed in *retrieve_backup()* (Figure 3.4).

The *name* and *password* buffers are read into from the user in *get_info()*. These are then concatenated into a string and passed to *system()*. This results in a command injection vulnerability, since the user input is never sanitized before passing it to

```
1  void retrieve_backup() {
2    char cmd[200];
3    get_info();
4    ...
5    snprintf(cmd, 200, "cat %s_%s.secure.bak", name, password);
6    system(cmd);
7  }
8
```

Figure 3.4: Snippet of *backup*'s Source Code.

system(). Hence, the user can execute arbitrary commands on the host operating system and read the *flag*.

3.2.2 Sampleak

sampleak is the service deployed on *teamvm3*, which allows users store and retrieve notes which are also stored as files, but unlike backup, the password is also stored in the note, so the user is required to provide a password when creating the note and needs to supply the correct password when retrieving them. The intended vulnerability is placed in *read_note()* (Figure 3.5).

The *password* buffer is read into by the user with *read()*. The length of *password* is 60 bytes but *read()* allows one to read in 200 bytes. This causes the *password* buffer to overflow and corrupt memory outside the buffer. If a carefully crafted payload is sent to the program, it can allow *remote code execution* on the host operating system and let the user read the *flag*.

```
1  static int read_note() {
2    char password[60];
3    ...
4    printf("Please send: password\n");
5    ...
6    length = read(0, & password, 200);
7    ...
8    return 0;
9  }
10
```

Figure 3.5: Snippet of *sampleak*'s Source Code.

3.2.3 Exploit-Market

exploit-market is the service deployed on *teamvm4*, which allows users to store, retrieve and list payload in the form of data which is stored in the memory of the program. The intended vulnerability is in *find_exploit()* (Figure 3.6).

The *payload* buffer is declared to be 500 bytes in *new_exploit()*, but in *find_exploit()*, it is declared to be only 200 bytes. So, when *strcpy()* executes on line 46, it can overflow the *payload* buffer in *find_exploit()*.

There is also an intentional bug placed in the service (Figure 3.7). This bug leaks memory addresses of *name*, *price*, and *payload* buffers for all the *exploits* objects created in *new_exploit()*. This bug makes it even easier for the user to exploit the service.

If a carefully crafted payload is sent in *new_exploit()*, it can overflow the *payload* buffer in *find_exploit()*, and allow *remote code execution* on the host operating system and let the user read the *flag*.

```
1  void new_exploit() {
2    char name[500];
3    char payload[500];
4    ...
5    scanf("%499s", name);
6    scanf("%499s", payload);
7  }
8
9  void find_exploit() {
10   char name[200];
11   char payload[200];
12   ...
13   strcpy(name, exploits[exploit_id].name);
14   strcpy(payload, exploits[exploit_id].payload);
15 }
16
```

Figure 3.6: Snippet of *exploit-market*'s Source Code.

3.3 Defense Mechanisms

Defense mechanisms are developed and deployed to protect the services from being exploited by the attacker. The two defense mechanisms being discussed are an intrusion detection system, specifically snort and the other one is honeypatching.

3.3.1 Deploying Snort

On the *ictf-framework*, snort has been deployed on the router machine. It sniffs the packets incoming from *teamvm1*, which is the machine used by the participant to exploit services running on the other *teamvms*. The rules filter used to setup snort is known as *indicator-shellcode.rules*, the reason for choosing this specific rules filter

```
#define DEBUG 1
void list_exploits() {
  if (DEBUG) {
    printf("%p\n", exploits[i].name);
  }
  if (DEBUG) {
    printf("%p\n", & exploits[i].price);
  }
  if (DEBUG) {
    printf("%p\n", exploits[i].payload);
  }
}

```

Figure 3.7: Snippet of *exploit-market*'s Source Code Of An Intentionally Placed Bug.

is because two out of the three services can be exploited by sending a *shellcode* as a payload when exploiting the buffer overflow vulnerability. On the other hand, it is extremely difficult to detect command injection accurately. Any kind of rules filter to detect exploitation of a command injection vulnerability would result in more false negatives than expected.

Snort is configured to block the connection from *teamvm1* to the *teamvm* which has snort setup as a defense mechanism for the service. It will block the connection for 180 seconds if a shellcode is detected in the payload, if it is also present in *indicator-shellcode.rules*.

3.3.2 Deploying Insider For Honey-patching

insider has been deployed inside the docker container running the vulnerable services. When the participant interacts with a vulnerable service via a tcp connection,

socat is listening for connections on the service host machine, which then executes a *spawn* python script, which is responsible for spawning a copy of the vulnerable service, pausing the execution of the program, and setting up the trampoline for the honey-patch to work as intended, as shown in steps 1-3 in Figure 2.1.

3.3.2.1 Honeypatch For Backup

backup has a command line injection vulnerability in *read_backup()*. The concatenated string *cmd* passed to *system()*, calls *get_info()*, which in turn calls another function *readline()* to read into the *name* and *password* buffers. This function can be used to setup the trampoline for the honey-patch.

Figure 3.8 shows a snippet of honey-patching *readline()*.

The patched *readline()* function acts as a command line injection filter while reading input, which essentially checks whether each character that is provided as input by the user can be considered dangerous (characters which can be potentially used to exploit a command line injection vulnerability). If it detects a dangerous character, the process sandboxes itself in a *chroot* environment.

3.3.2.2 Honeypatch For Sampleak

sampleak has a buffer overflow vulnerability in *read_note()*. The *password* buffer can be overflowed, so a trampoline function has to be setup after reading into the *password* buffer. This function is named *is_easy_password()* in the original program and placed before returning from the *read_note()*.

Figure 3.9 shows a snippet of honey-patching *is_easy_password()*.

The patched *is_easy_password()* function checks if the length of the payload is greater than 60 bytes, which is the original size of the *password* buffer. If the check fails, the process sandboxes itself in a *chroot* environment.

3.3.2.3 Honeypatch For Exploit-Market

exploit-market has a buffer overflow vulnerability in *find_exploit()*. The *name* and *password* buffers can be overflowed, so a trampoline function has to be setup between reading into the vulnerable buffers and the *strcpy()* which causes the overflow. This function is named *is_bad()* in the original program.

Figure 3.10 shows a snippet of honey-patching *is_bad()*.

The patch for *is_bad()* works similarly to the honey-patch for *is_easy_password()* from *sampleak*. The subtle difference here is that there are more than one vulnerable buffers in *exploit-market*, and also 200 bytes in size.

```
int _p_readline(char * buf, int size) {
  for (...) {
    if (read(0, buf, 1) <= 0) {
      exit(1);
    }

    if ((buf[i] == ' ') ||
      (buf[i] == '!') ||
      (buf[i] == '$') ||
      (buf[i] == '%') ||
      (buf[i] == '&') ||
      (buf[i] == '(') ||
      (buf[i] == ')') ||
      (buf[i] == '|') ||
      (buf[i] == ';') ||
      (buf[i] == '<') ||
      (buf[i] == '>') ||
      (buf[i] == '?') ||
      (buf[i] == '/')) {
        // sandbox the process using chroot
      }
      ...
    }

```

Figure 3.8: Snippet of Honey-Patching *readline()*.

```
int length;
int _p_is_easy_password(char * password) {
  if (length > 60) {
    // sandbox the process using chroot
  }
  ...
}

```

Figure 3.9: Snippet of Honey-Patching *is_easy_password()*.

```
int _p_is_bad(char * shellcode) {
  int length = strlen(shellcode);
  if (length > 200) {
    // sandbox the process using chroot
  }
  ...
}

```

Figure 3.10: Snippet of Honey-Patching *is_bad()*.

Chapter 4

EXPERIMENTAL DESIGN

Once the infrastructure is setup, a standard procedure has to be decided to conduct experiments involving human participants. The standard procedure involves several steps to make sure the experiments are in a stardized manner. Starting off with, before every experiment starts, the setup of defense mitigations is randomly selected for each service. The possible mitigations include, snort, honey-patch or even no mitigation at all. The service-mitigation relationship is one-to-one. Next step is to find a willing participant and a time is prescheduled to meet for the experiment on an online meeting platform, *Zoom*. Then, the participant goes through the consent document and if they agree, the goal of the experiment is explained to them, which is to exploit the service and read the contents of */flag* and send it over in *Zoom* chat before time runs out. Each challenge has a fixed alloted time to exploit. After that, participant's *SSH public key* is added to the *teamvm1*'s *authorized_keys* file, and they are given access to *teamvm1* by providing them the *public IP* of the machine to *SSH* into. Once they *SSH* into the *teamvm1* machine, the directory structure of the services is explained to them and right after they can start hacking! The timer is started and they move from one challenge to the other, in the order *backup*, *sampleak*, *exploit-market*, until they successfully exploit the service (read and send */flag* in *Zoom* chat) or the time runs out, whichever occurs first. Once the experiment concludes, participant is debriefed about the defenses, and their feedback is taken for each challenge and also if their questions are answered, if they have any. This helps with making changes to the infrastructure and tweaking things a bit in the procedures of the experiments which are discussed in the next chapter. The last step is to gather

data from all the *teamvms* which is collected using various tools like *tcpdump* and *SysFlow*. *tcpdump* is deployed on the router machine and captures all the network traffic sent and received between *teamvm1* and other *teamvm* machines. Next, all the data collected by *SysFlow* which is deployed on all the *teamvms* is fetched. Finally, collect the *syslog* data which is deployed on *teamvms* running the vulnerable services. *syslog* is setup to collect debug and error messages on the *teamvms* for further analysis, in case something goes wrong during the experiments.

Chapter 5

RESULTS

The experimentation started with a pilot study. In the pilot study, the participant is able to exploit only one service successfully. In this experiment, the honey-patch is visible to the participant as it is deployed inside the binary of the service. An unintended format string vulnerability is also found in *sampleak*. The participant, thus, tries to exploit the unintended vulnerability to bypass the honey-patch deployed on *sampleak*. Changes are implemented for the upcoming experiment.

This time the unintended vulnerability is removed and a transparent honey-patch (not visible to the participant directly) is deployed. But the honey-patch has been made intentionally bypassable using a *null byte* in the exploit payload. A couple more experiments are conducted, with more changes like giving the participants access to the source code of the services and letting them know about where the vulnerabilities are present in the source code, since most of their time is spent on vulnerability analysis instead which is not the goal of this work.

In one of the experiments, the participant is able to unknowingly bypass the honey-patch for *sampleak*, this results into implementing a non-bypassable honey-patch. Due to so many changes being made in the experiments, the above experiments are neglected when coming up with the final conclusion. In response to the feedback given by the participants, appropriate changes are made.

Until this point, all the major changes and implementations have been completed, so all the experiments conducted afterwards are taken into account when coming up with the final conclusion.

Figure 5.1 shows the average time taken by the participants to exploit each chal-

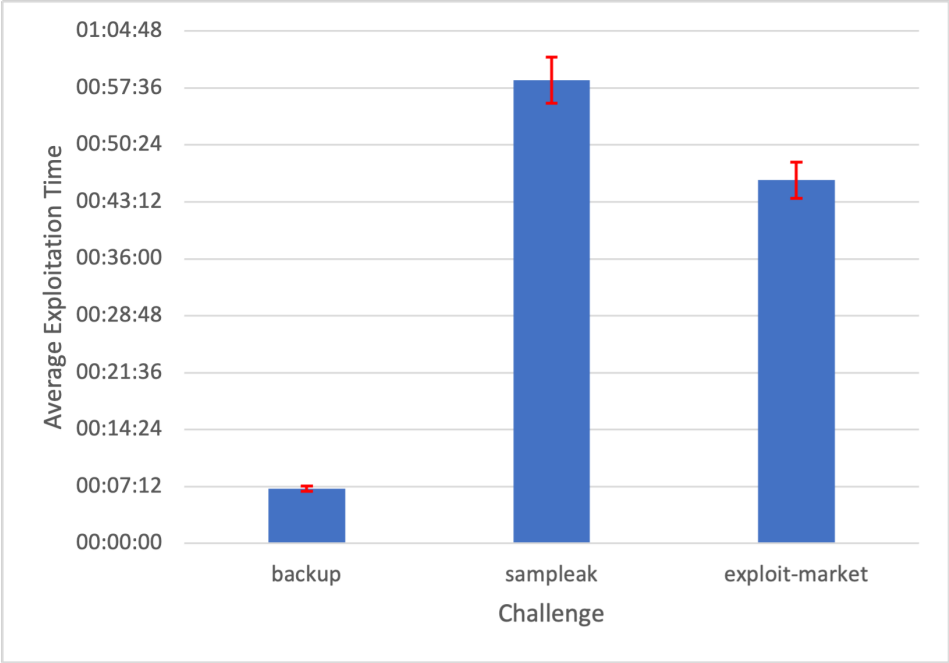

Figure 5.1: Average Exploitation Time Per Challenge.

		Exploitation with Defense Mechanisms			
Challenge	**Successful Exploitation**	**None**	**Snort**	**Honeypatch**	**Gave Honeypot Flag**
backup	**18/18**	**6/6**	**6/6**	**6/6**	**6/6**
sampleak	**9/18**	**4/6**	**2/5**	**3/7**	**3/3**
exploit-market	**9/15**	**3/5**	**3/5**	**3/5**	**3/3**

Table 5.1: Summary Of ICTF Experiments.

lenge. This statistic excludes the participants who timed out on a challenge. For 95% confidence, the error amount percentage is set to 5%. This figure signifies that participants found it most difficult to exploit *sampleak*, followed by *exploit-market*, then *backup*.

Table 5.1 shows the summary of the experiments conducted. Here, the number of successful attempts for each experiment has been represented as a fraction of the total number of experiments. Each row corresponds to a different service and columns represent the various exploitation scenarios.

Chapter 6

DISCUSSION

Looking at the results, total 24 experiments were conducted over a time period of 11 months. They kept evolving so things had to be changed as they progressed. Towards the end, 6 different experiments were neglected since several major changes were implemented due to the feedback received from the participants.

Several interesting observations were noted after analyzing the results, the most important observation was whenever honey-patch was deployed as a mitigation and the participant was able to successfully exploit that service with the mitigation, they always gave the *honeypot flag*. A solid conclusion cannot be inferred from this, but an assumption could be made that the particiapant, due to having a time constraint, always gave away the first *flag* they saw. It can also be assumed that given enough time they could have escaped the *honeypot* and found out the real *flag*. Some unique instances of special circumstances that occured are discussed ahead.

During some of the experiments, after the participants got to know about the honey-patches, they were able to successfully escape the *chroot* sandbox and read the real */flag*. In one instance, the participant was clearly able to know that they were in a sandbox environment, but due to a lack of time, they gave the fake honeypot *flag* so they could move onto the next challenge. But this is due to the way the *chroot* sandboxes are designed, they are vulnerable to sandbox escape exploits if certain conditions are met. One way to make honeypots more resilient is to use more advanced sandboxing tools like *SELinux Sandbox* [24] or *Virtual machines* [22]. But even then, zero-day vulnerabilities are very common nowadays for those tools which can still make it possible for attackers to escape sandboxes setup using honey-patches.

It has also been observed during analysis of honey-patched programs, that if an attacker has achieved arbitrary code execution in the program, they can use *return oriented programming (ROP)* techniques to detect the presence of a honey-patch by dumping the program instructions and then, analying them by reversing machine code.

Another observation is that when honey-patches are deployed on vulnerable programs, they affect the execution time due to more instructions (code) needing to be executed. Attackers can notice this timing difference and get more vigilant when performing attacks.

Chapter 7

RELATED WORK

7.1 Studies On Game-Theoretic Approaches In Cybersecurity

Several studies have been done before which involves game-theoretic approaches to study or solve various problems faced in the field of cyber security [2, 6–10, 12, 15, 18, 19, 23, 29, 30, 33, 34].

An interesting survey, *Game Theory for Cyber Security and Privacy* [8] explores known game-theoretic techniques for cyber security and privacy issues. It highlights the advantages and limitations from the design to implemenation of defense systems along with game models, features and solutions for the selected works. It not only exhibits how to use game theory to address security and privacy issues, but also motivates researchers to utilize game theory to gain a solid understanding of evolving security and privacy issues in cyberspace, along with viable solutions.

For more than two decades, the field of network defense mechanism has gotten a lot of interest from the research community, but due to the complexity and the difficulty of network security, this issue is far from being solved. *A Survey of Game Theory as Applied to Network Security* [19] examines existing game theoretic approaches aimed at improving network security and proposes a taxonomy for categorizing the solutions to have a better insight into these approaches to solve a variety of cyber security concerns.

7.2 Previous Work On Conducting Cyber Security Experiments

Cyber security exercises have several uses, they can be used as a platform to teach cyber security concepts or also to conduct experiments to study, analyze and solve issues related to cyber security [1, 11, 14, 16, 20, 21, 25–27].

One very interesting work, *Cyber Security Exercises and Competitions as a Platform for Cyber Security Experiments* [26] discusses the use of cyber security exercises and competitions to obtain data vital for security research. Generally, cyber security exercises and competitions are organized to train and/or test proficiency of participants with a passion in security. This paper explores how exercises and competitions in the field of security can be utilized as a foundation for experimentation.

Chapter 8

CONCLUSION

Upon analyzing the results it is a clear indication honey-patching a vulnerable program always results in an attacker getting trapped into the *chroot* sandbox and giving the fake honeypot flag. But it seems too good to be true. On further analysis of unique scenarios which occured in certain experiments, it seems that honey-patches are not a perfect security solution.

The experiments conducted gives a good amount of data to analyze which can have statisctical signifance, when running game-theoretic models to prove the effectiveness of various cyber defense strategies like honey-patches. These game theoretic models could provide a solid conclusion to find the most effective security mitigation.

Although a preliminary conclusion can be made after analyzing the results, taking into account the special cases in certain experiments and interesting observations that honey-patching is a viable security mitigation due to it being easy to setup and when deployed alongside modern mitigations like Data Execution Prevention (DEP), Address Space Layer Randomization (ASLR), and many more, it can turn out to be beneficial. But it also has its downsides such as, increased overhead and being potentially detected by an attacker.

REFERENCES

[1] Aljohani, A. and J. Jones, "Conducting malicious cybersecurity experiments on crowdsourcing platforms", in "The 2021 3rd International Conference on Big Data Engineering", pp. 150–161 (2021).

[2] Alpcan, T. and T. Başar, *Network security: A decision and game-theoretic approach* (Cambridge University Press, 2010).

[3] Araujo, F., K. W. Hamlen, S. Biedermann and S. Katzenbeisser, "From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation", in "21st ACM Conference on Computer and Communications Security (CCS'14)", (ACM, 2014).

[4] Araujo, F. and T. Taylor, "Improving cybersecurity hygiene through jit patching", in "Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering", pp. 1421–1432 (2020).

[5] Attiah, A., M. Chatterjee and C. C. Zou, "A game theoretic approach to model cyber attack and defense strategies", in "2018 IEEE International Conference on Communications (ICC)", pp. 1–7 (2018).

[6] Attiah, A., M. Chatterjee and C. C. Zou, "A game theoretic approach to model cyber attack and defense strategies", in "2018 IEEE International Conference on Communications (ICC)", pp. 1–7 (IEEE, 2018).

[7] Dasgupta, P. and J. Collins, "A survey of game theoretic approaches for adversarial machine learning in cybersecurity tasks", AI Magazine **40**, 2, 31–43 (2019).

[8] Do, C. T., N. H. Tran, C. Hong, C. A. Kamhoua, K. A. Kwiat, E. Blasch, S. Ren, N. Pissinou and S. S. Iyengar, "Game theory for cyber security and privacy", ACM Computing Surveys (CSUR) **50**, 2, 1–37 (2017).

[9] Hasan, S., A. Dubey, G. Karsai and X. Koutsoukos, "A game-theoretic approach for power systems defense against dynamic cyber-attacks", International Journal of Electrical Power & Energy Systems **115**, 105432 (2020).

[10] Kamhoua, C., A. Martin, D. K. Tosh, K. A. Kwiat, C. Heitzenrater and S. Sengupta, "Cyber-threats information sharing in cloud computing: A game theoretic approach", in "2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing", pp. 382–389 (IEEE, 2015).

[11] Kavak, H., J. J. Padilla, D. Vernon-Bido, R. Gore and S. Diallo, "A characterization of cybersecurity simulation scenarios.", in "SpringSim (CNS)", p. 3 (2016).

[12] Kiennert, C., Z. Ismail, H. Debar and J. Leneutre, "A survey on game-theoretic approaches for intrusion detection and response optimization", ACM Computing Surveys (CSUR) **51**, 5, 1–31 (2018).

[13] Luo, Y., F. Szidarovszky, Y. Al-Nashif and S. Hariri, "Game theory based network security", J. Information Security **1**, 41–44 (2010).

[14] Mäses, S., K. Kikerpill, K. Jüristo and O. Maennel, "Mixed methods research approach and experimental procedure for measuring human factors in cybersecurity using phishing simulations", in "18th European Conference on Research Methodology for Business and Management Studies", p. 218 (2019).

[15] Merrick, K., M. Hardhienata, K. Shafi and J. Hu, "A survey of game theoretic approaches to modelling decision-making in information warfare scenarios", Future Internet **8**, 3, 34 (2016).

[16] Mirkovic, J. and T. Benzel, "Teaching cybersecurity with deterlab", IEEE Security & Privacy **10**, 1, 73–76 (2012).

[17] Mitchell, R. and B. Healy, "A game theoretic model of computer network exploitation campaigns", in "2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)", pp. 431–438 (2018).

[18] Musman, S. and A. Turner, "A game theoretic approach to cyber security risk management", The Journal of Defense Modeling and Simulation **15**, 2, 127–146 (2018).

[19] Roy, S., C. Ellis, S. Shiva, D. Dasgupta, V. Shandilya and Q. Wu, "A survey of game theory as applied to network security", in "2010 43rd Hawaii International Conference on System Sciences", pp. 1–10 (2010).

[20] Salah, K., M. Hammoud and S. Zeadally, "Teaching cybersecurity using the cloud", IEEE Transactions on Learning Technologies **8**, 4, 383–392 (2015).

[21] Salem, M. B. and S. J. Stolfo, "On the design and execution of {Cyber-Security} user studies: Methodology, challenges, and lessons learned", in "4th Workshop on Cyber Security Experimentation and Test (CSET 11)", (2011).

[22] Santhanam, S., P. Elango, A. Arpaci-Dusseau and M. Livny, "Deploying virtual machines as sandboxes for the grid", pp. 7–12 (2005).

[23] Schlenker, A., O. Thakoor, H. Xu, M. Tambe, P. Vayanos, F. Fang, L. Tran-Thanh and Y. Vorobeychik, "Deceiving cyber adversaries: A game theoretic approach", in "International Conference on Autonomous Agents and Multiagent Systems", (2018).

[24] Schreuders, Z. C., T. J. McGill and C. N. Payne, "The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls", Comput. Secur. **32**, 219–241 (2013).

[25] Schwab, S. and E. Kline, "Cybersecurity experimentation at program scale: Guidelines and principles for future testbeds", in "2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)", pp. 94–102 (IEEE, 2019).

[26] Sommestad, T. and J. Hallberg, "Cyber security exercises and competitions as a platform for cyber security experiments", in "Nordic conference on secure IT systems", pp. 47–60 (Springer, 2012).

[27] Stransky, C., Y. Acar, D. C. Nguyen, D. Wermke, D. Kim, E. M. Redmiles, M. Backes, S. Garfinkel, M. L. Mazurek and S. Fahl, "Lessons learned from using an online platform to conduct {Large-Scale}, online controlled security experiments with software developers", in "10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)", (2017).

[28] Taylor, T., F. Araujo and X. Shu, "Towards an open format for scalable system telemetry", in "2020 IEEE International Conference on Big Data (Big Data)", pp. 1031–1040 (IEEE Computer Society, Los Alamitos, CA, USA, 2020), URL https://doi.ieeecomputersociety.org/10.1109/BigData50022.2020.9378294.

[29] Thakkar, A., S. Badsha and S. Sengupta, "Game theoretic approach applied in cybersecurity information exchange framework", in "2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)", pp. 1–7 (IEEE, 2020).

[30] Tom, L., "Game-theoretic approach towards network security: A review", in "2015 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015]", pp. 1–4 (IEEE, 2015).

[31] Trickel, E., F. Disperati, E. Gustafson, F. Kalantari, M. Mabey, N. Tiwari, Y. Safaei, A. Doupé and G. Vigna, "Shell we play a game? CTF-as-a-service for security education", in "2017 USENIX Workshop on Advances in Security Education (ASE 17)", (USENIX Association, Vancouver, BC, 2017), URL https://www.usenix.org/conference/ase17/workshop-program/presentation/trickel.

[32] Vigna, G., K. Borgolte, J. Corbetta, A. Doupe, Y. Fratantonio, L. Invernizzi, D. Kirat and Y. Shoshitaishvili, "Ten years of ictf: The good, the bad, and the ugly", (2014), funding Information: This work was supported by the National Science Foundation, through grants CNS-0820907, CNS-0716753, and CNS-0939188, and by the ARO through MURI grant W911NF-09-1-0553. We want to especially thank Carl Landwehr, Jeremy Epstein, and Karl Levitt at the National Science Foundation for their support to cyber-competitions. Publisher Copyright: © 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education, 3GSE 2014. All rights reserved.; 2014 USENIX Summit on Gaming, Games, and Gamification in Security Education, 3GSE 2014 ; Conference date: 18-08-2014.

[33] Wang, Y., Y. Wang, J. Liu, Z. Huang and P. Xie, "A survey of game theoretic methods for cyber security", in "2016 IEEE First International Conference on Data Science in Cyberspace (DSC)", pp. 631–636 (IEEE, 2016).

[34] Zhao, Y., L. Huang, C. Smidts and Q. Zhu, "A game theoretic approach for responding to cyber-attacks on nuclear power plants", Nuclear Science and Engineering (2021).