Development of an Offensive Cybersecurity Curriculum

Jacob Mills

Department of Computer Science

Florida State University

Supervisor

Dr. Mike Burmester

In partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

May 3, 2019

Acknowledgements

The completion of this work would not have been possible without the guidance and support of many people whose names could not all be mentioned here. Their contributions are priceless, greatly appreciated, and sincerely acknowledged. However, I would like to particularly acknowledge the following:

Dr. Xiuwen Liu, Dr. Mike Burmester, and Dr. Randolf Langley of the Florida State University Department of Computer Science for their countless contributions and inspiration in the classroom. You have all helped to create environments in which learning is both fulfilling and rewarding.

To all my friends and family for your continual support and understanding as I have endeavored to achieve greatness in academia. You are the ones whom keep me going nights on end.

To my peers and contemporaries whom I've had the privilege to study alongside. You have shown me the many facets of expertise involving computer science and cybersecurity and I have aspired to learn from each of you.

Abstract

In 2008 The Congressional Research Service provided a report to Congress suggesting that advanced cybersecurity skills are available for hire, potentially to nation states or terrorist organizations [1]. Threats posed by cyber mercenaries have grown in intensity, frequency, and severity over the past decade, and academia has been slow to produce a highly skilled workforce to combat modern cyber operations. Some experts speculate that this may be related to the lack of curricula specific to applied cybersecurity, as the subject is often presented as a special topics course for computer science students, which may result in the under-representation of cybersecurity principles and strategies [2]. In this paper I present the development of an offensive cybersecurity curriculum. The following material is the basis of the implementation of the Practical Cyber Operations Fundamentals course of which I taught at Florida State University in Spring 2019. It is my goal to both educate students regarding hands-on cyber operations at an advanced level while also providing the framework of a curriculum that can be later reimplemented by aspiring faculty or graduate students.

Contents

1 Introduction **1**

1.1 Motivations 1

1.2 Previous Works 2

1.3 Objectives and Contributions 3

1.4 Course Organization 5

1.4.1 Prerequisite Coursework 6

1.5 Project Deliverables 7

2 Systems Architecture **8**

2.1 Host Requirements 9

2.2 Server Requirements 9

2.2.1 Capture The Flag Framework 10

2.2.2 Containerization 11

2.2.2.1 Hosting Python Programs 13

2.2.2.2 Hosting Web Applications 14

2.2.2.3 Hosting C Programs 16

3 Course Content **18**

3.1 Scripting 19

3.2 Web Application Exploitation 20

3.3 Digital Forensics 23
3.4 Fundamentals of the C Programming Language 25
3.5 Software Reverse Engineering 27
3.6 Binary Exploitation 32
3.7 Cryptography 41

4 Capture The Flag Events **45**

5 Conclusions **47**

A Term Group Project **48**

B Assignment Creation Template **54**

C Assignment Submission Template **57**

D CTFd docker-compose.yml **59**

E /etc/nginx/sites-enabled/CTFd **61**

F Sample Python Dockerfile **62**

G LEMP docker-compose.yml **63**

H Sample xinetd Configuration **65**

References **67**

Chapter 1

Introduction

This paper presents the development of an offensive cybersecurity curriculum which was implemented in the Spring semester of 2019 at Florida State University in the form of Special Topics in Computer Science courses, CIS4930 and CIS5930, Practical Cyber Operations Fundamentals, offered at both the undergraduate and graduate levels, respectively. The material in this work is comprised of the collective efforts of myself and others whom strove to strengthen the quality of education related to applied cybersecurity at Florida State University.

1.1 Motivations

As I progressed through my career in academia I received multiple opportunities in the technical workforce for positions such as application developer, cybersecurity analyst, and software reverse engineer. While working in these positions I began to notice a divergence from the theoretical and mathematical standards taught in academia when compared to the daily activities of technical professionals. It became clear to me that while academia was driven by theoretical models established through time, there was not enough focus on real-world problems

to adequately prepare the cyber workforce of tomorrow. Dishearteningly, this particularly seemed to apply to cybersecurity. For example, students may be tasked to create web applications as part of a course project in database theory, but vulnerabilities in the applications they created were never regarded. This results in poor practices that transfer over to the industry through unaware and unprepared technical professionals.

On the other hand, the technical industry is driven by budgets, timelines, and executives that may not know about, or be interested in, cybersecurity. This often leads to carelessness or impatience when developers write code. As a result, unknown vulnerabilities may appear in applications that are prominent in the marketplace. It is my goal to increase the practicality of the skills acquired through computer science related coursework by providing hands-on experience involving several topics related to present-day cybersecurity. To do so I've designed a curriculum that focuses on applicability over theory, understanding over memorization, and critical review over rapid completion.

1.2 Previous Works

The material presented in this paper is largely a derivative work based on the contributions of several before me. Spring 2019 was the third semester that the Practical Cyber Operations Fundamentals course was taught at Florida State University. Although the name of the course has slightly changed over the past two years, along with the instructors, the methodology of the course remains primarily unaltered. I took the course in its original form and have since taught it twice. I intend to improve upon the foundations of the previous curricula by both enhancing the training material provided to students as well as increasing the documentation and resources available to potential future instructors.

1.3 Objectives and Contributions

Upon successful completion of this course of study, the student will gain proficiency in:

- Developing scripts for solving various problems related to cybersecurity
- Identifying and exploiting common web application vulnerabilities
- Utilizing SQL injection techniques to exploit vulnerable database driven applications
- Analyzing common file formats (ELF, PE, DOCX, PDF, and more)
- Recovering hidden or deleted information from disk images
- Analyzing binary programs in x86 assembly
- Identifying and exploiting buffer overflow vulnerabilities in binary executables
- Identifying and exploiting string format vulnerabilities in binary executables
- Developing and using shellcode as a binary exploitation technique
- Understanding the fundamentals of return-oriented programming (ROP) and return-to-libc (ret2libc) attacks
- Recognizing common weaknesses in implementations of cryptographic algorithms
- Performing cryptanalysis of substitution and commonly used ciphers

To meet these objectives the curriculum must cover several disciplines related to cybersecurity. The primary categories representing each discipline are covered in

greater detail in the Course Organization section. My primary contributions to this work include:

- Designing and implementing the system architecture of a web server used to host a centralized and interactive assignment submission framework along with individual, containerized, challenges of which the homework assignments are comprised of
- Organizing the course structure in an effort to meet all objectives within a single semester
- Creating lecture materials and homework assignments for the Scripting, Web Exploitation, Reverse Engineering, and Binary Exploitation categories
- Assisting students through lectures, workshops, emails, and office hours
- Coordinating efforts between FSU computer science faculty and teaching assistants
- Documenting the process of implementing the curriculum

I must acknowledge the contributions made by instructors of previous versions of this course - in particular, Shawn Stone for his endeavors to standardize containerization, binary analysis, and binary exploitation, Douglas Hennenfent for his diligent documentation of forensics exercises, and Brandon Everhart for his insights on reverse engineering. Additionally, I would like to recognize my primary teaching assistants for the Spring 2019 semester, Jordan Mussman and Hannah McLaughlin, for their contributions on the Forensics and Cryptography portions of the course, respectively. Finally, I would like to thank Dr. Xiuwen Liu and Dr. Mike Burmester for making the realization of this curriculum possible in the form of a special topics course for computer science.

1.4 Course Organization

Cybersecurity is an increasingly broad field with a large variety of disciplines and demands. Therefore, the content of this curriculum is organized in the following categories in order to optimize the learning experience for students:

- Scripting
- Web Exploitation
- Digital Forensics
- Fundamentals of the C Programming Language
- Software Reverse Engineering
- Binary Exploitation
- Cryptography

Each of the aforementioned categories are covered for a period of two weeks, with the exception of the Fundamentals of the C Programming Language and Software Reverse Engineering categories, which are each covered within a week and a half. Assignments are given weekly in the form of Capture The Flag (CTF) challenges related to the primary subject of that week. The assignments are graded based on the quality of student write-ups describing the analysis, strategies, and solutions required for solving each CTF problem. An assignment submission template is provided to students to help facilitate standardization in student submissions (see Appendix C). CTF problems are popular amongst the offensive cybersecurity community because they require a great deal of problem solving skills along with technical prowess. A supplementary goal of this course is to increase the CTF presence of Florida State University in national and international competitions.

The remaining weeks of the course are used for the practice CTF and final CTF events, which are required and constitute a significant portion of a student's final grade. A term project has been added to the course to engage students in exercises involving the creation of CTF challenges (see Appendix A). This project is mandatory for graduate students and an opportunity for extra credit for undergraduates. There are no quizzes nor exams in this course.

Syllabi for the undergraduate and graduate versions of this course are not included within this document, but they will be included in the archive of files submitted with this research project, along with homework assignments and configuration files required for creating the server system architecture (see the Project Deliverables section for more information). The syllabi outline in greater detail the grading scales and tentative schedules for each course derived from this curriculum. They should serve as a resource for any additional references to course organization.

1.4.1 Prerequisite Coursework

Historically, the subjects in this curriculum that students tend to struggle with the most are Reverse Engineering and Binary Exploitation. Both of these subjects rely on a strong background in instruction set architectures and assembly languages. Therefore, CDA 3100 - Computer Organization and Design I is a prerequisite course. Students are expected to have a strong programming background as well as proficiency with UNIX derivative operating systems. Although courses exist for learning Python, reverse engineering, and cryptography, the completion of those courses does not appear to be directly essential for success in this class. This course is intended to be an elective for senior or graduate students in computer science or a related field of study. I have recently considered that an ethics course, particularly ethics as applied to technology, should be an ad-

ditional prerequisite. This consideration is based on the notion that the skills acquired throughout this course have ethical implications and it is illegal to use them in certain contexts. I will leave it to the discretion of future instructors as to whether or not the prerequisite coursework should be altered.

1.5 Project Deliverables

This document primarily serves as a formal description of the development of an offensive cybersecurity curriculum. Due to the nature of this endeavor, supplemental materials are required for such a curriculum to be implemented. The complete set of deliverables for this project will include the following:

- This document
- Syllabi from courses created to implement the material described in this paper
- Lectures and assignments associated with this curriculum
- Templates and other supplemental documentation
- Binary files of which assignments are comprised of
- An archive containing the necessary documents for creating the server system architecture for hosting assignments

Upon the review of these deliverables the recreation of this work should be possible.

Chapter 2

Systems Architecture

Summary

One of the major challenges in implementing the proposed curriculum is establishing an environment in which offensive cybersecurity skills can be practiced. The techniques that students learn could result in serious legal repercussions if practiced on real-world systems. However, it is my goal to provide an opportunity for students to gain hands-on experience with penetration testing. Therefore, an environment must be created where students have permission to demonstrate their mastery of the course material. This chapter intends to provide an overview of both the environment that students are expected to utilize throughout their study as well as the systems architecture required to establish realistic scenarios in which the exploitation of cyber systems results in minimal damages and optimal learning experiences.

2.1 Host Requirements

Host requirements refers to the environment that students should be using throughout their study. For this class, it is recommended that students use a Unix derivative operating system. One of the most popular Linux distributions for generic use is currently Ubuntu. It offers an intuitive user interface along with a command line terminal suitable for all tasks associated with this course. A virtual machine (VM) of the Ubuntu operating system has been preconfigured at http://www.sait.fsu.edu/sait-usb/vms/CTF_Ubuntu.ova. The majority of software required throughout this course have been preinstalled on the virtual machine available at the previous link. Any additional software that may be useful for completing assignments will be mentioned in the lecture material. Although certain academic departments provide Linux environments for student use, it is not advised to use them for this curriculum, as administrative privileges will be necessary for installing and utilizing some of the software discussed throughout the course.

2.2 Server Requirements

One of the most complicated requirements for creating the systems architecture for this course is establishing an accessible server to host resources for students, including an interactive assignment submission platform and containerized services that can be exploited without compromising the infrastructure of the server. I used https://www.digitalocean.com/ to create an Ubuntu Linux server with the necessary resources for this task. My particular instance included 80 gigabytes of storage, 4 gigabytes of RAM, and 2 virtual CPUs. This ended up costing $20 per month of hosting. I used nginx as the primary web application framework on the server. I decided to purchase a domain for the server so students are not required

to memorize an IP address. I purchased the domain fsuctf.com from GoDaddy for $12 a year. The domain choice should be simple and reflect the course, but keep in mind that some domain names cost more than others (less expensive ones costing around $12 annually). After configuring the server to utilize the purchased domain and adjusting any nameservers accordingly, a SSL certificate can be added to the server to help facilitate HTTPS connections. I used the certificate authority https://letsencrypt.org to acquire a free SSL certificate for my domain. Within a terminal connected to the server, the CertBot utility can be used to automate the majority of retrieving a security certificate and binding it to nginx. Finally, steps are required to create the Capture The Flag framework and support containerization.

2.2.1 Capture The Flag Framework

A Capture The Flag Framework is essentially a web application for users to interact with by registering, logging-in, accessing challenges, and submitting solutions. Solutions take the form of a *flag*, which is a textual token of completion acquired after appropriately solving individual challenges. Although *flags* are sufficient for receiving credit for the completion of challenges in formal CTF competitions, it is customary for experienced users to submit write-ups describing their process for solving various problems and demonstrating their mastery of the subject material. Therefore, the Capture The Flag Framework for this course is strictly intended to be a platform for students to access problems and verify their solutions (in the form of flags). Students are required to submit write-ups through Canvas and the quality of write-ups are evaluated to determine the grades that students receive for each assignment.

A popular CTF framework in the cybersecurity community is CTFd, available at https://github.com/CTFd/CTFd. A number of options for hosting CTFd

are available, including uwsgi, Flask, and gunicorn. However, for consistency I decided to use Docker to host CTFd in a containerized environment. Doing so requires mapping an external port on the server to an internal port on the Docker container, which will be elaborated in the Containerization subsection. Since I host other services on my DigitalOcean server, I used port 8000 for both the external and internal ports of the CTFd instance. Additionally, the CTFd Docker instance requires a mariadb instance for database operations along with a redis instance to provide caching services. Docker instances can be managed via the docker and docker-compose utilities. Although each utility includes pros and cons, it is generally a good practice to manage containers with multiple services through docker-compose. This includes the CTFd container. A complete docker-compose.yml file for the CTFd container is listed in Appendix D for reference. Additional steps are required to associate the CTFd Docker instance with the domain used by the server. This association can be facilitated through nginx, which can be used as a simple HTTPS proxy to redirect traffic destined to https://fsuctf.com over port 443 to the Docker listener at port 8000. The /etc/nginx/site-enabled/CTFd entry performing this proxy service is provided in Appendix E.

2.2.2 Containerization

The Docker documentation states that a container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, run-time, system tools, system libraries and settings [3]. A Docker instance is an image that is actively running on the Docker engine. Therefore, a single image can have zero to many instances, as long as

each instance is uniquely identified. The parallel is similar to that of a process and a program. Containerization provides a few benefits over virtualization, primarily concerning resources. The idea is that containers share and reuse many of the resources provided by the host operating system that is running the Docker engine, whereas virtual machines must allocate memory for an entire operating system and map emulated hardware to resources on the host.

Throughout this curriculum containerization will be used to host programs on an accessible web server in an isolated environment. Since exploiting vulnerabilities in these programs is aligned with our learning objectives, it is imperative that these programs are isolated to mitigate the potential to compromise the host system. This is one of the fundamental challenges of implementing an offensive cybersecurity curriculum - creating vulnerable environments for training and supporting them with a secured environment for hosting. Fortunately, Docker is a lightweight solution to this challenge if configured correctly. There are three principle types of applications which need to be hosted in isolated containers: Python, Web, and C. For each type of application different approaches to containerization have been considered. All hosted applications should be accessible via an Internet connection. The DigitalOcean server, domain name, nginx, and Docker facilitate primary accessibility. Additionally, internal Docker instances should have the capability of serving multiple clients. Although Docker provides a mapping from external to internal ports, an internal service is required to accept incoming connections and service multiple clients concurrently. Each program could be written to handle socket connections; however, this creates extra work when writing programs to serve as CTF challenges, diverges from standardization in deployment, and altogether creates inconsistencies since it is common for CTF challenges to be written by multiple authors. For these reasons, I decided to use a daemon for managing Internet-based services from inside a docker instance. In

particular, the applications xinetd and tcpserver were used. These daemons not only perform the required tasks for accepting connections and serving multiple clients concurrently, but they also copy the file descriptor of a socket to standard input and standard output of the respective instance of an executing program. Therefore, hosted programs can be written in such a way that utilizes standard input and standard output for I/O operations without regards to networking semantics. Note that in some cases firewall exceptions may be required to enable external connections to Docker instances.

2.2.2.1 Hosting Python Programs

Python programs are fairly straightforward to host in a Docker container. Because of this, I decided to use tcpserver as the server daemon. First, you will need to create a Python program for your CTF challenge. I/O behaves differently over network sockets, so be certain to include the Python system library with `import sys` and flush standard output with `sys.stdout.flush()`. This will force tcpserver to write standard output to the client explicitly; otherwise, data may be stored in the standard output buffer longer than what is desired. Another best practice is to include CTF flags in separate documents than the challenge itself. This prevents accidentally printing a flag as part of a stack trace if an error occurs during execution. After a Python program has been prepared, you will need to create a Docker image that runs it. This can be accomplished by preparing a Dockerfile for the specific program. Appendix F includes an example Dockerfile for a Python program named `challenge.py` which is hosted internally on port 31337 by tcpserver. A simple script can be used to deploy the program to Docker. Note that the first two commands in the following script will stop the Docker instance and remove the Docker image, the third command builds an image according to the Dockerfile in the current directory, and the final command

creates a new instance of the Docker image named *challenge* as a background daemon, names the instance *challenge*, and maps the external server port 2201 to the internal instance port 31337.

```
#!/bin/bash
docker kill challenge
docker rm challenge
sudo docker build -t "challenge" .
docker run -d --name=challenge -p 0.0.0.0:2201:31337 challenge
```

It should be noted that internal ports are independent across Docker instances, but external ports cannot be reused. Based on our Python Dockerfile, the internal instance is using tcpserver to host `challenge.py` on port 31337. Everything described thus far can be reused to host other Python programs with the exception of the program name, instance name, image name, and port. I encourage anyone recreating this work to write scripts to automate as much of the deployment process as possible.

2.2.2.2 Hosting Web Applications

Web applications require a HTTP service to handle client connections and interface with any server scripting languages that are responsible for producing dynamically generated web pages. A common server scripting language for CTFs is PHP. PHP is a powerful language that is easy to configure and free to use. However, PHP is highly unforgiving of programming errors introduced by developers. The improper implementation of PHP may result in vulnerabilities in an application allowing for the interaction with database management systems in an unauthorized manner, exfiltration of files from the web server, manipulation of content that is served by the web application, or even remote code execution.

When hosting a PHP application in Docker it is necessary to create an image supporting a HTTP server, PHP, and potentially a backend database. Nginx is a lightweight HTTP service and for that reason it is a good candidate for dockerization. Another benefit of using a HTTP service such as nginx is that it is already designed to facilitate connections with multiple clients, so an additional network daemon such as tcpserver is not required. The goal is to create the Linux Nginx MySQL PHP stack (LEMP) within a Docker container. Such a container can be used to host one or more web CTF challenges. Note that PHP applications require a `php.ini` file specifying the configuration of the PHP process manager. Minor modifications to the php.ini file may be necessary to allow for the exploitation of certain vulnerabilities included in a PHP application.

Appendix G provides a `docker-compose.yml` file to create a Docker image with services for nginx, phpfpm, mysql, and phpmyadmin. This image serves as the LEMP stack and can host challenges involving PHP applications. The `docker-compose` utility can be used to deploy an instance of this container as a daemon background process. If required, database connectivity is facilitated through the `mysqli` PHP library. The easiest way to create and administer databases in an instance of this image is through the phpmyadmin portal, which provides a graphical user interface as well as a MySQL command line.

Challenges involving exploitation in the form of local file inclusion (LFI) or remote code execution (RCE) should be hosted in individual containers, where as most other web challenges can be hosted on a single container. It is often easier to configure images for such containers through a standalone Dockerfile, as opposed to utilizing `docker-compose`. Building and deploying these containers is similar to what was covered in the Hosting Python Programs subsection. Below is a simple bash script that can be used to automate the deployment process for simple web Docker images.

```
#!/bin/bash

if ! [ "$#" -eq 3 ]
then
    echo "Usage: $0 image name port"
    exit 1
fi

docker kill $2
docker rm $2
docker build --file .docker/Dockerfile -t $1 .
docker run --name $2 --rm -p $3:80 $1
exit 0
```

2.2.2.3 Hosting C Programs

Binary exploitation challenges commonly involve hosting compiled C programs within Docker containers. The process is similar to hosting a Python program. Instead of tcpserver, I used xinetd to host C programs due to is advanced configuration options and versatility. This process is more tedious than configuring tcpserver, but it works well and allows for turning on and off system protection mechanisms such as address space layout randomization (ASLR). The basic process is outlined as follows:

- Create a simple bash script called `init.sh` that executes a generic binary name, such as `prob.bin`. This script may invoke the `setarch` command to disable ASLR, if desired.
- Create a xinetd configuration file specifying the internal port to listen to and

the program to execute upon successfully accepting an incoming connection, which in our case is `init.sh` (see Appendix H).

- Create a Dockerfile that fully supports the required architecture for running a compiled C binary, as well as xinetd. This Dockerfile should expose the appropriate internal port as well as execute a script that starts the xinetd service.
- Build the docker image
- Deploy an instance of the image and map it to an accessible external port. Don't forget to copy the challenge binary to the instance as `prob.bin` and create a flag.txt file on the instance.

A bash script for automating the deployment of an image called ctf-default32 is provided below for reference:

```
#!/bin/bash
usage="Usage: ./deploy.sh <binary> <port> <instance_name> <flag>"
if [ "$#" -eq 4 ]
then
    sudo docker run -d --privileged --name $3 -p \
    "0.0.0.0:$2:9999" ctf-default32
    sudo docker cp $1 $3:/home/ctf/prob.bin
    sudo docker exec $3 chmod +x /home/ctf/prob.bin
    echo $4 > temp_flag
    sudo docker cp temp_flag $3:/home/ctf/flag.txt
else
    echo $usage
fi
```

Chapter 3

Course Content

Summary

In order to meet the diverse objectives for this curriculum, content has been aggregated into a set of overlapping disciplines within the field of offensive cyber security. Although these disciplines need not be covered sequentially, they have been arranged such that the skills developed throughout the study of one subject carries forward to the next. That being said, from my experience it is helpful for students to become familiar with the basics of scripting and file analysis before attempting reverse engineering or binary exploitation. Furthermore, the following arrangement of course topics tends to present material with increasing difficulty. In the following sections a subject will be presented and the primary learning objectives for that subject will be discussed. Throughout a semester each subject will be covered for a period of one to three weeks.

3.1 Scripting

Scripting and automation is an essential discipline throughout any field of modern computing. Compilation and interpretation increasingly provide an abstraction from the mechanical nature of instruction set architectures and central processing units. This allows computer users to develop algorithms and solve problems more efficiently with respect to time and complexity. Fluency with the basics of Unix shells and bash scripting is expected of students. However, alternative scripting environments are often helpful when solving problems related to cyber security. A natural choice is Python, which has been growing in popularity over the past several years. I have found that the learning curve for writing basic Python scripts is quite gentle, and students with prior programming experience should have no issues learning Python in a short period of time. Additional benefits of the Python language include extensibility and portability. This course is not intended to provide an in-depth background in the object-oriented features of Python. Instead, this portion of the curriculum focuses on quickly solving problems in an understandable and uncomplicated fashion.

Since students are not required to have prior experience with Python per the prerequisites, the basic features and syntax of the language deserves a lecture or two during the first week of the course. Students should acquire familiarity with type usage, functions, lists, loops, and conditional expressions. It is also important to discuss the basics of I/O operations and file manipulation, such as reading from and writing to files. Simple algorithms should be discussed and practiced in class to reinforce language features and syntax usage.

In Capture The Flag events scripting skills are often useful for brute forcing inputs to programs, interacting with binaries using either static or dynamic inputs, interacting with files on a system, or interacting with services over a network. Thus, it is important to include a lecture discussing advanced techniques

and networking in Python. This lecture should provide insights for converting types between binary, integer, hexadecimal, ordinate, character, and string formats. Additionally, students should gain experience with concepts for list comprehensions, formatting techniques, map and reduce functions, substrings, range functions, encodings, and the `itertools` Python library for string generation. Although itertools can be useful to exhaustively brute force inputs to a program, students should also gain experience with dictionary attacks.

The networking component of this section can be covered using a variety of Python libraries. The `telnetlib` and `socket` libraries are likely the most intuitive for beginners, so they are presented here. Later in the curriculum the `pwntools` library will be introduced, which provides an even more versatile way to interact with services over a network. Students should understand the basics of connecting to a remote host through a listening port and performing I/O operations on the underlying socket. The homework challenges for this section involve performing tedious, albeit simple, operations on a remote host through a network connection. It should be infeasible for students to complete these challenges by hand. Therefore, students should be writing scripts to connect to network services, parse data, and provide the appropriate inputs to successfully interact with a remote application throughout a long series of dynamic events.

3.2 Web Application Exploitation

Over the past several years dynamic web applications have become increasingly popular. Unfortunately, developers continue to write vulnerable code in these applications, and sometimes even web server frameworks. This is likely attributed to time constraints, budget limitations, lack of appropriate quality assurance practices, and unfamiliarity with secure application development. The primary goal

of this section is to educate students on the nature of websites, application layer protocols, popular encodings, and common vulnerabilities found in web applications. Furthermore, upon completion of this part of the curriculum students should be prepared to exploit vulnerabilities in real-world web applications as part of quality assurance auditing or penetration testing.

Before students begin learning about vulnerabilities in web applications, they must first understand the basics of how websites works. They should be presented the Hypertext Transfer Protocol (HTTP) and what HTTP headers look like for both requests and responses. They should also understand that a web application is accessible through a web service that listens to an inbound port and maps successful connections to files designed to serve either static or dynamic content. Typically, network connections for web applications are facilitated through ports 80 or 443 (depending on if HTTPS is used), although a web service can be bound to any unused port through proper configuration. HTTP request methods, particularly GET and POST, should be explained to students along with how to pass parameters for each respective method. The concept of HTTP cookies and session management should also be discussed, which will prepare students for understanding the fundamentals of session hijacking. Lastly, web encodings such as URL encoding and base64 should be presented and students should become familiar with converting data to and from encoded forms.

After covering the essentials of HTTP, web applications, and web clients, students will be prepared to utilize command line utilities for web interactions, such as curl, wget, netcat, and telnet. They should also be presented the `requests` library for sending HTTP requests in Python, which will be useful for scripting communications with websites, and additional libraries such as `BeautifulSoup` and `PyQuery`, which are helpful for parsing HTML. Now that students understand how websites are supposed to work, they can start to develop an understanding

of how websites can be exploited. I find this to be a useful time for explaining the differences between static and dynamic pages, along with naming a few popular web servers, such as nginx, apache, and IIS. Students should become familiar with changing parameters passed through HTTP requests to potentially cause runtime errors on web servers or enable access to data that may not be intentionally exposed.

Students are now prepared for learning SQL injections. First, the ANSI SQL standard and the syntax of SQL in various database management systems (DBMS) should be reviewed. Next, examples of SQL injections through parameter manipulation should be demonstrated. There are a number of resources for legally practicing these techniques, many of which are included in the lecture slides and source files supplemental to this document, but instructors are always encouraged to create their own vulnerable applications for demonstrations and assignments. Next, techniques for local file inclusion (LFI), cross-site scripting (XSS), and remote code execution (RCE) should be discussed in detail, along with defense strategies against all vulnerabilities presented in this section.

The advanced portion of this section should present techniques for LFI using PHP-specific wrappers, SQL injection using UNION statements or the SQLmap utility, remote code execution used in conjunction with LFI exploits, server-side injection, and cross-site request forgery (CSRF). Upon completion of this material students should understand how to find common vulnerabilities in web applications, how to exploit these vulnerabilities using various techniques, and how to prevent the vulnerabilities through cybersecurity awareness and implementation of best practices for secure application development.

3.3 Digital Forensics

Digital forensics operations involves the analysis of digital data in the form of files, network traffic, or even memory states. This is a growing field with applications in research, law enforcement investigations, penetration testing, and more. Before each of these areas can be further explored, it is important to establish a foundation of data encodings. Perhaps the most import question to present to students is, *what is data*? For our purposes, data is information encoded in a binary form. This information may represent numbers, characters of an alphabet, machine language instructions, addresses in memory, or any combination thereof. It should be stressed that although sequences of binary digits may be grouped together at variable lengths before translation, the standard unit of digital information is a byte, or eight binary digits. The goal of this section is to provide students with experience in digital forensics operations involving system, network, and memory analysis.

The first step of understanding digital forensics is establishing a foundation in the fundamentals of data encodings. As previously mentioned, all digital data assumes the form of a binary encoding. This data can be translated into a variety of contexts. Popular encodings for textual data include ASCII and Unicode. Numeric data is typically translated from a binary form into either base16 hexadecimal or base10 decimal values. In the Web Application Exploitation section base64 encoding was introduced and it is still relevant for digital forensics. Other encodings may involve cryptographic techniques such as hashes, ciphers, and xor encryption with 2-4 byte keys, although the distinction between encodings and ciphers is not necessarily established until the Cryptography section. To aid students it may be beneficial to introduce the CyberChef website, which accepts input and provides translation to and from a large variety of popular data encodings.

After learning about different data encodings students should be prepared to perform basic file analysis. They should be introduced to file signatures, or *magic bytes*, that can be used to identify the content of files. The `file` command is a useful utility for classifying file types based on filesystem tests, file signature tests, and language tests. The `strings` command can be used to identify sequences of bytes in a file that are within the printable range of characters, and the `hexdump` or `xxd` commands are handy for viewing the bytes of a file in a hexadecimal encoding. All of these commands are essential for file analysis and can be used to detect file types, obfuscation, or hidden content in binary data. Additionally, students will undoubtedly encounter file archives in the form of `.tar`, `.bzip2`, or `.7z` formats and they should be trained to condense and extract data from each archive type, respectively. Finally, students should become familiar with file carving with tools such as `binwalk` and `foremost`, comparing files with the `diff` and `cmp` commands, and analyzing file metadata with `exif` and `exiftool`.

Network forensics involves the analysis of data that is transferred through a network medium, such as Ethernet or WiFi. A few popular tools for network forensics are `Wireshark` and `tcpdump`. The essentials of network communications presented in Web Exploitation should be built upon to discuss the data link layer, the network Layer, TCP/UDP fundamentals, and application layer protocols beyond HTTP. Students should understand how MAC and IP addresses are used for switching and routing network packets, along with the functionality of ARP, DHCP, and DNS protocols. With this background students should be prepared to analyze packet captures, identify conversations between network hosts, and sort traffic by protocol, addresses, timestamps or any combination of filters. Advanced practices in network forensics should involve decrypting SSL and TLS traffic by using a key file.

Memory forensics is the third and final category of forensics discussed in this

section. It may be difficult to develop assignments involving memory forensics due to the large size of memory captures. `Volatility` is a useful tool for performing memory forensics. Examples of extracting data from running processes and files included in memory should be demonstrated for students. A useful exercise would be to have students extract the NTLM hashes from a memory dump of a Windows system, then use a password cracker such as `John The Ripper` to determine the passwords on the system.

3.4 Fundamentals of the C Programming Language

The title of this section is a bit of a misnomer. This curriculum does not intend to cover the fundamentals of the C programming language, as it is assumed that students have prior experience writing code in C (or minimally C++). Instead, this section aims to provide a background in understanding and exploiting common vulnerabilities found in programs written in the C language. C is a powerful language because it interacts very succinctly with hardware in the form of machine language instructions, memory management, and system calls. Unlike interpreted languages, such as Java, Python, or C#, C does not execute within a virtual runtime environment. Rather, C source files are compiled into a target assembly language supported by the central processing unit (CPU) of a particular machine. This section aims to guide students through understanding the processes of compilation and assembly as applied to the C programming language. These skills will greatly assist students with reverse engineering and binary exploitation, which are covered later in the curriculum.

Fundamental concepts involving the C programming language that students should first review are format strings, I/O operations, and library functions. The

`printf`, `sprintf`, and `scanf` functions should be understood by students, along with their respective format specifiers. A union in C is a special data structure that allows for storing different data types at a shared location in memory. Familiarity with unions is often helpful for understanding memory management and viewing data through various contexts, which parallels the notion of format string specifiers. Concepts related to memory allocation, local variables, and global variables help students understand the multiple sections in virtual memory during the execution of a binary. Finally, Linux system calls are essential for understanding how user space applications interact with the operating system kernel.

As Computer Organization and Design is a prerequisite to this curriculum, students are expected to have a background with the MIPS assembly language. Although this course will focus primarily on the x86 assembly language, a refresher in MIPS can help students prepare for a more thorough analysis of the compilation and assembly of programs in the C language. Students should be presented with strategies for designing a rudimentary MIPS disassembler. Afterwards, these concepts can be expanded to describe how to create a disassembler for x86. At this time, basic x86 instructions should be presented to students, along with instruction formats, encodings, and disassembly techniques.

One of the primary vulnerabilities in C programs involves the incorrect usage of format strings. In short, if a user can control a format specifier, data can be read from or written to arbitrary, or specific, locations in memory. Printing data in various contexts through the usage of format strings should be presented to students, followed by format string vulnerabilities. Students should become familiar with the layout of stack frames to better understand how stack variables and return addresses can be viewed or manipulated through format string exploitation. Exercises should be designed and presented to task students with

mastering these concepts.

The second vulnerability of interest within this section involve buffer overflows. The stack frame should be revisited and students should understand what happens when data is written beyond the bounds of a buffer. The results may be anything from overwriting the values of other stack variables to altering the value of a function return address. With the knowledge acquired so far, students will be prepared for the discussion of control flow exploitation through code reuse and shellcode attacks. Exploitation involving code reuse will generally take the form of overwriting the return address of a function to that of another valid sequence of instructions. Exercises should be given to students tasking them to execute a particular function within a program through the use of buffer overflow exploitation. Although students will not receive exercises involving shellcode until the Binary Exploitation section, they should be presented with the concept of shellcode, how it can be generated, and the process of executing shellcode through a buffer overflow exploit.

3.5 Software Reverse Engineering

Software reverse engineering refers to the study and analysis of binary executables for the purpose of better understanding their design, operation, and development. In practice this can be used to learn more about particular software where documentation or source files may not be available, produce improved or competitive versions of software, or identify vulnerabilities within software so that exploits can be crafted accordingly. Additional applications include the analysis of malicious software or legacy systems and the validation of software quality and robustness. This section focuses on understanding the binary life cycle, techniques binary analysis, and identification of vulnerabilities within binary executables.

Before students are prepared to perform binary analysis, they must first be introduced to a variety of helpful utilities for binary disassembly. A few basic disassemblers include `objdump` and `gdb`, although they are both very intuitive and helpful in simple scenarios. A more advanced resource for disassembling binary executables is `radare2`, which offers a useful graphical view of the control flow of disassembled instructions. Over the past several years `IDA Pro` has been the industry standard disassembler, offering an intuitive graphical interface for viewing sections of a binary, control flow graphs, a debugger, and even a decompiler for users with an advanced software license. Unfortunately, the full version of `IDA Pro` is expensive and the trial version is limited in functionality. For this reason it is not an ideal resource for students. Another popular disassembler offering similar functionality to `IDA Pro` is `Binary Ninja`, which is also unfortunately unavailable for free. Recently, the National Security Agency (NSA) released a previously propriety disassembler known as `Ghidra` to the public. `Ghidra` is both free and open source, making it a great candidate for this curriculum. Students are encouraged to utilize `objdump`, `gdb`, `radare2`, or `Ghidra` for performing binary analysis throughout this course.

To aid students with understanding software reverse engineering it is important to review the binary life cycle. Students should become familiar with the concepts of preprocessing source files written in textual encodings such as ASCII, compiling preprocessed files into instructions for a target assembly language, assembling assembly files into machine language object files, linking object files to form a binary executable, and finally loading a binary executable to create an image of the running program in memory. Reverse engineering tends to focus on working backwards through the binary life cycle to retrieve assembly instructions equivalent to the original compilation result. Due to loss of abstraction throughout the process of compilation, the retrieval of original source files is not feasible.

However, it is useful to describe the challenges of decompilation to students and introduce them to freely available decompilers such as `Ghidra`.

After the binary life cycle has been reviewed students will be prepared to learn the essentials of the x86 assembly language. This curriculum concentrates primarily on x86 over over alternatives such as MIPS or ARM because of the prevalence of x86 architectures in modern laptop and desktop computers. However, this section should introduce the concept of using hardware emulation to execute binaries for alternative instruction sets by using tools such as `qemu`. Students should be able to compile binaries in `C` or `C++`, analyze the disassembly using one of the tools mentioned in this section, and begin to parallel how the algorithms written in high-level programming languages are realized through machine instructions. Although an understanding of all x86 instructions is not required, students should learn the fundamental instructions for moving data, loading addresses, performing arithmetic, interacting with the stack through push and pop operations, evaluating conditional statements, and calling functions. Next, various x86 calling conventions should be discussed with students to address issues related to passing parameters, managing the stack, sharing registers, and returning data. Finally, the memory sections of an ELF binary should be described to students to help them understand the virtual address space of a running program. In particular, the `.text`, `.data`, `.heap`, and `.stack` sections and their usage should be explained.

At this point, students should be prepared to perform static analysis of ELF binaries using command-line tools such as `file`, `strings`, `readelf`, `objdump`, and `radare2`. An essential part of static analysis involves examining the control flow of disassembly. Some utilities, such as `radare2`, offer a graphical view of the control flow in a binary. This is helpful for viewing sequences of instructions as basic blocks, or groups of instructions that execute sequentially until a branch oc-

curs. The understanding of basic blocks and branch conditions will help students when analyzing the control flow of binary executables to determine how particular inputs or conditions can cause different paths of execution. Then, students should be prepared to identify how logical constructs such as functions, loops, conditionals, and switch statements are implemented in assembly languages.

The next step is for students to learn techniques for dynamic analysis, which involves examining the state of a binary during execution. The most fundamental technique for performing dynamic analysis is the use of a debugger. Although several debuggers exist, this curriculum will mostly focus on `gdb` and `gdb peda`, which is a python extension of `gdb`. Students should learn how to both debug binary executables explicitly and attach debuggers to running processes. Additionally, the use of breakpoints to temporarily halt the execution of a process should be understood, and students should become fluent with setting breakpoints, examining register and stack values, and stepping through instructions incrementally. Also, advanced debugging techniques such as changing the value of registers or data on the stack should be understood. Other important techniques to cover related to dynamic analysis involve process tracing, file access, and network activity monitoring.

The challenges of decompilation should be revisited and students should review why it is difficult to produce the equivalent source code of binaries compiled in languages such as C. This is a useful time to discuss the difference between compiled languages and interpreted languages. Interpreted languages are often highly object-oriented, resulting in several binary artifacts to implement concepts such as reflection. These languages are generally so abstract that nearly every instruction utilizes a class. Because of this, references to memory are strongly typed and as a result it is fairly straightforward to retrieve the original source code from a language compiled into an intermediate form (Java, C#, Python,

etc). To illustrate this concept, examples of decompiling a Java class can be shown using a decompiler such as `jad`.

A common technique for reverse engineering involves fuzzing, or generating a large amount of random input of variable size and sending it to a program until undefined behavior occurs. If input resulting in undefined behavior can be identified, dynamic analysis can be performed to determine potential vulnerabilities in a binary and where to look for them. A reasonable source for dynamic randomized input on a Linux system is `/dev/urandom`. Since this course covers interacting with binaries through Python, it is also useful to present Python fuzzers such as `zzuf`.

Up to now the primary focus of software reverse engineering has been on how code at a source level is translated into a machine form and the efforts of reversing the translation. It is time to introduce some basic concepts of linked libraries and their utility. The dynamic loader is responsible for resolving library calls at runtime. However, this loader can be preloaded to overwrite standard library functions with those of your own. One such technique is known as `LD_PRELOAD`. The process of compiling functions as shared, position independent, object files and overwriting function definitions using the `LD_PRELOAD` technique should be described to students. An extension of this topic involves function hooking, which tends to intercept function calls before and after their execution instead of entirely replacing them. These techniques are useful for the concept of binary instrumentation, which involves the guiding of a process control flow during execution through manipulation of inputs, outputs, resources, addresses, or more. If students are interested in learning more about binary instrumentation I recommend sharing the `Frida` project.

The final advanced topic for software reverse engineering is symbolic execution. Symbolic execution is a means of analyzing a program to determine what

inputs cause each part of the program to execute. It is an attempt to combine the strengths of static and dynamic analysis, offering theoretically perfect code coverage. Since the code is simulated, it is possible to identify complex vulnerabilities that are difficult to find statically. Students should be presented with basic concepts of symbolic execution along with popular symbolic execution engines such as `angr`.

3.6 Binary Exploitation

Binary exploitation is the process of subverting an application such that it violates some trust boundary in a way that is advantageous an attacker. This may involve modifying or patching the binary before runtime, instrumenting the binary during runtime, or introducing inputs that the application fails to sanitize appropriately, causing unanticipated runtime behavior from the perspective of the developer. In general, the goal of binary exploitation is to change the value of the instruction pointer during the execution of a program. Two techniques involving modification of the instruction pointer are discussed in this section:

- Executing instructions that are already included in the binary
- Executing instructions that are dynamically written into the address space of a process during execution

Before further exploration of the art of binary exploitation, it may be useful to review a few previously discussed concepts. Students should fully understand the endianness of a system and how it determines the interpretation of data. A common misunderstanding in the difference of little and big endian is that the bits are stored in reverse, which is untrue. The endianness of a system specifies the logical grouping of bytes for types that occupy more than one byte. The

Executable and Linkable Format (ELF) should also be reviewed. Students must understand the ELF program header table and memory sections of a binary. At this time symbol tables should be introduced along with the concept of stripped binaries. Students should be comfortable with binary analysis techniques using the various utilities presented in the Software Reverse Engineering section. Finally, it is immensely important for students to have a thorough understanding of the stack frame and its use. For our purposes, binary exploitation relies on successful manipulation of the stack frame. An understanding of stack concepts will aid students in crafting payloads to accomplish exploitation.

This section presents techniques for exploiting binaries on remote machines through a network connection. This is the primary distinction that this curriculum designates between software reverse engineering and binary exploitation. As a super user, it usually isn't very exciting to exploit binaries running on your own machine, unless you're attempting to instrument or crack software. In practice, the exploitation of remote binaries may result in the compromising of external systems. The CTF category that requires techniques consistent with our definition of binary exploitation is known as `pwn`. The goal of `pwn` challenges is usually to access a shell, perhaps even a root shell, on a remote system. Typically, `pwn` challenges provide a host address and a port. A copy of the program running on the remote machine along with any dynamically linked libraries may also be provided.

The primary utility for binary exploitation presented in this section is `pwntools`. `Pwntools` is a collection of tools and python libraries intended to make the life of a penetration tester easier. It provides a CTF framework and exploit development library, making scripting exploits through Python as easy as possible. `Pwntools` provides command line utilities for checking binary protection mechanisms, classes for interacting with local processes and remote applications, ELF

parsing, packing, shellcode and ROP generation, and much more. Students should be given instructions on installing pwntools along with example code for generating shellcode and interacting with local or remote programs. Since pwntools includes more features than time allows for covering in this curriculum, students should be instructed to visit https://docs.pwntools.com/en/stable/ and refer to the documentation when necessary.

Before `pwntools` can be used effectively, students must understand particular vulnerabilities within binaries and how they can be exploited. The first such vulnerability is known as a buffer overflow. In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. With an understanding of the stack frame, students should see how buffer overflows can result in overwriting local variables, stack canaries, return addresses, parameters, and more. The concepts behind buffer overflow exploitation should already clear after the Fundamentals of the C Programming Language section. However, it may be useful to review how a buffer overflow vulnerability such as `strcpy` can be used to overwrite the return address of a function, result in calling another function within a binary. This is a form of code reuse attacks. Other code reuse attacks involve ROP chains or overwriting the Global Offset Table. These techniques will be discussed in greater detail later in this section.

Shellcode is a sequence of bytes that represent valid instructions for a particular instruction set architecture. Shellcode is used to open a shell, which is generally done with the machine code equivalent of `execve("/bin/sh",NULL)` in C source. Students should understand why shellcode works and how it can be generated using `pwntools`. Then, students should be able to analyze a binary, identify any buffer overflow vulnerabilities, write shellcode onto the stack,

and perform a buffer overflow to overwrite the return address of a function to the address of the shellcode on the stack. This process may involve leaking or retrieving the address of an input buffer, calculating the amount of padding required to successfully overwrite the return address, and bypassing any binary protection mechanisms that may be present. Finally, students should be prepared to script binary exploitation using Python and `pwntools` to acquire a shell on a vulnerable remote system.

As an attempt to mitigate binary exploitation, a number of binary protection mechanisms have been introduced over the past several years. Students should be aware of these mechanisms, what they do, and how to defeat them. The first mechanism to discuss is Position Independent Code (PIC). Position Independent Executables (PIE) are an output of the hardened package build process. A PIE binary and all of its dependencies are loaded into random locations within virtual memory each time the application is executed. This makes Return Oriented Programming (ROP) attacks much more difficult to execute reliably. However, binaries implementing PIC can still be exploited by leaking data relative to addresses an attacker may be interested in, such as a return address or local buffer. Position Independent Executables are also found to be ineffective in systems that do not implement Address Space Layout Randomization (ASLR).

Address space layout randomization (ASLR) is a memory-protection process for operating systems that guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory. On Linux this feature can be temporarily disabled through the `setarch `uname -m` -R ./bin` command, where `./bin` is the name of the binary to execute. When it is desirable for ASLR to be disabled through a session, it may be useful to replace `./bin` with `/bin/bash`. For a more permanent solution, ASLR can also be configured through the `/proc/sys/kernel/randomize_va_space` file. The following values of that

file are supported:

- 0 - No randomization. Everything is static.
- 1 - Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized. written into the address space of a process during execution
- 2 - Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized.

Stack canaries are randomly generated values placed on the stack frame during the prologue of each function call. During the function epilogue, the value that was placed on the stack is checked against the original value, and an exception routine executes when the values do not match. This is problematic for buffer overflow exploitation, because overwriting the return address requires first overwriting the stack canary. This technique can be defeated by leaking the value of the canary before sending the final payload, which preserves the canary and overwrites the return address.

Another binary protection mechanism is known as the NX bit, or write-xor-execute (W$\oplus$X). This technique marks segments within the binary address space as read, write, or execute. Combinations of read, write, and execute are allowed, with the exception of write and execute. Shellcode will not work on binaries implementing this technique, but code reuse attacks will.

The final binary protection mechanism to discuss is known as Relocation Read-Only (RELRO). By default binaries tend to perform lazy-loading, which involves only mapping the addresses of dynamically linked symbols when they are first invoked. This topic requires prior understanding of the Global Offset Table, Procedure Linkage Table, and Dynamic Linking. Therefore, RELRO will be revisited after a discussion on linking.

The process of compilation converts source files into object files. Each object file contains the machine code instructions corresponding to the statements and declarations in the source program. The symbols of an object file can be viewed using a utility such as `nm`. The symbol table will list each symbol in an ELF binary, the section the symbol is located in, and the address or offset to reach the symbol from the start of that section. However, symbols from external libraries have undefined sections and offsets. The resolution of these symbols requires a dynamic linker. The linker's primary function is to bind symbolic names to memory addresses. A static library is an archive structure that stores a collection of raw object files strung together along with a table of contents for fast symbol access. When a static library is included during program linking, the linker makes a pass through the library and adds all the code and data corresponding to symbols used in the source program. Although static libraries are easy to create and use, they present a number of software maintenance and resource utilization problems. For example:

- When the linker includes a static library in a program, it copies data from the library to the target program. If patching the library is ever necessary, everything linked against that library must be rebuilt for the changes to take effect.

- Copying library contents into the target program wastes disk space and memory - especially for commonly used libraries such as the C library.

To address the maintenance and resource problems with static libraries, most modern systems use shared libraries or dynamically linked libraries (DLLs). The primary difference between static and shared libraries is that using shared libraries delays the actual task of linking to runtime, where it is performed by a special dynamic linker-loader. To facilitate this task it is helpful to define a table of

relocation structures. Then, redirection can be used to lookup the addresses of functions that are needed during runtime. In ELF binaries this table is known as the Procedure Linkable Table (PLT). The PLT is a table of stub functions, one for each dynamically loaded function, plus some extra stubs. Each function stub contains an entry to the Global Offset Table (GOT) where the address of the actual function has been dynamically loaded. If the GOT entry for a function has not yet been populated, control is returned to the PLT where a special routine is executed to invoke the dynamic loader and update the GOT. A second call to the GOT entry is then made, which this time will result in redirecting to the dynamically loaded symbol. Lazy loading defers the resolution of dynamically linked symbols until they are needed, which does offer performance benefits. However, to implement lazy loading the GOT must be writable. If a vulnerability exists within a binary allowing an attacker to overwrite the GOT entry for a symbol, the address of either shellcode or an existing function could be written in the place of a dynamic symbol address. Whenever that symbol is subsequently accessed, the attacker's instructions would then be executed instead of the original function. Immediate loading is an alternative to lazy loader which forces the dynamic loader to load the addresses of shared functions at program startup. Full RELRO uses immediately loading and then marks the GOT as read-only. Partial RELRO uses lazy loading, but marks certain sections as read-only after they have been initialized by the dynamic loader.

The implementation of the Standard C library as described in C standards is known as `libc`. It is common for all compiled C programs to include `libc` as a shared object. The inclusion of `libc` to use one function, such as `printf` or `__libc_start_main` actually includes the entire `libc` object file. Therefore, other functions within `libc`, such as `system`, are included in virtually every binary compiled from C source. This lends itself useful for code reuse attacks, as

binaries may already include symbols to perform system calls in `libc`. An attacker can leverage this by simulating the call to `system("/bin/sh")` through binary exploitation. To accomplish this task, it is necessary to know the address that `libc` is dynamically loaded at. In systems utilizing ASLR this may be a problem. However, if data can be leaked from the address space, the base address of the `libc` shared object can be calculated, and a payload can be carefully crafted to call symbols within `libc`. Two popular code reuse attacks that perform such an exploit are known as `return-to-libc` (ret2libc) and `return oriented programming` (ROP).

This section describes two techniques that can be used to leak data from the address space of a process, buffer overflow leaks and format string leaks. Consider the following C source code:

```
int main(int argc, char *argv[]) {
    char buf[20];
    int i;
    for (i = 0; i < 10; i++) {
        read(0, buf, 40);
        printf("%s", buf);
    }
    return 0;
}
```

In the above code it should be clear that reading over 20 bytes of data from standard input will result in overflowing the `buf` character array. For each iteration of the loop, the value of the buffer is printed by the `printf` function, which will continue to print bytes until the "null byte" is encountered. This code can be exploited to overwrite the null byte of the stack canary (if present), leaking that

value, and sending a final payload which preserves the stack canary and overwrites the return address of the function. This is an example of a buffer overflow in a loop, which is most common in programs with vulnerabilities in repetitive menu functions. An example exploit that could be used to leak the value of the canary would be to write 24 A's followed by a newline, as illustrated in Table 3.1.

Table 3.1: Buffer Overflow Leaks

Before Overwrite			After Overwrite		
rbp-0x20	0x7fffffffd460	0x400650	rbp-0x20	0x7fffffffd460	AAAAAAAA
	0x7fffffffd468	0x4004e0		0x7fffffffd468	AAAAAAAA
	0x7fffffffd470	0x7fffffffd560		0x7fffffffd470	AAAAAAAA
Canary	0x7fffffffd478	0x6b5c162ef30f4e00	Canary	0x7fffffffd478	0x6b5c162ef30f4e0a
rbp	0x7fffffffd480	0x400650	rbp	0x7fffffffd480	0x400650
return address	0x7fffffffd488	0x7ffff7a2d830	return address	0x7fffffffd488	0x7ffff7a2d830

Another way to leak data is through the exploitation of a format string vulnerability. Format strings such as `%s`, `%x`, `%d`, and `%llx` can be used within a payload to leak data from the stack. A sequence of these format specifiers, such as `%llx.%llx.%llx.%llx.%llx` can be used to leak adjacent data. Even better, direct parameter access, such as `%11$llx`, can be used to leak data at a specific offset from the expected format string location on the stack, which in this case would be the 11th 8-byte decimal value. Binary analysis should be used to determine the appropriate offsets to stack values of interest, then specific values such as a stack canary or the return address of main can be leaked using these techniques.

The final topic involving binary exploitation is return oriented programming (ROP). ROP is a type of code reuse attack involving the collection of valid instruction sequences in a manner that executes existing code with a malicious result. ROP can be used to defeat most mitigation techniques mentioned so far. A ROP Gadget is a small instruction sequence ending with the `ret` instruction. A ROP chain is a sequence of ROP Gadgets. The typical malicious ROP chain has the following structure:

- Address of ROP Gadget `pop rdi; ret;`
- Address of string "/bin/sh"
- Address of `system` function

If the addresses of the items listed above can be calculated through analysis or leaking data, a ROP chain can be crafted. If the return address of a function is overwritten with the ROP chain, the gadget `pop rdi; ret;` will execute, causing a pointer to the string "/bin/sh" to be removed from the stack and stored in register `rdi`, and the `system` function will be called with "/bin/sh" as an argument. On most modern Linux systems, `/bin/sh` is an alias for bash or another default shell. Therefore, this technique will cause the exploited binary to open a shell, if successful. Students should have a fundamental understanding of exploiting binaries through crafting ROP payloads. Exercises in exploiting binaries with various protection mechanisms enabled should be given to students as assignments.

3.7 Cryptography

Cryptography is the study of sending secured information through insecure communication channels. Secure cryptographic systems should provide services for confidentiality, authentication, integrity, and non-repudiation. In this section various concepts related to modern cryptography are presented, including terminology and algorithms. Students should learn the definitions of plaintext, ciphertext, encryption, hashing, and oracles with respect to cryptographic systems. Additionally, students should become familiar with common attacks on cryptosystems, including chosen-plaintext attacks, chosen-ciphertext attacks, frequency analysis, and brute-force attacks. The performance of different cryptographic algorithms

should be analyzed, along with methods to defeat these algorithms and the feasibility of doing so successfully.

Students should start with a refresher in modular arithmetic for bitwise operations. This will aid students in understanding cryptographic algorithms that utilize functions for performing a bitwise exclusive OR on input data. Then, simple substitution ciphers should be presented, such as the Caesar and Atbash ciphers. Exercises for encrypting and decrypting data using these methods should be demonstrated in class or as part of coursework. A slightly more complicated substitution cipher is the Vigenére cipher, which uses a key to shift each letter in a message by an amount determined by the key vector. It should be clear that a Vigenére cipher using a key of length 1 is equivalent to a Caesar cipher. Although the Vigenére cipher demands additional complexity when decrypting a message without the key, students should learn how frequency analysis can be used on the output of any substitution cipher to probabilistically determine the corresponding plaintext of a given ciphertext much more efficiently than exhausting all possibilities.

Perfect Secrecy means that the ciphertext conveys no information about the content of the plaintext. In effect this means that, no matter how much ciphertext you have, it does not convey anything about what the plaintext and key were. This is only possible if the key size is greater than or equal to the message size, so that the key vector does not repeat itself. This is the idea of a one-time pad. To encrypt a message $\boldsymbol{M}$ with length $\boldsymbol{n}$, the sender generates a secret key $\boldsymbol{k}$ such that $\mid \boldsymbol{k} \mid \geq \boldsymbol{n}$. Then we have ciphertext $\mathbf{C} = \boldsymbol{M} \oplus \boldsymbol{k}$ for each message block $\boldsymbol{M}$ and plaintext $\boldsymbol{M} = \mathbf{C} \oplus \boldsymbol{k}$ for each ciphertext block $\mathbf{C}$. The security of a one-time pad requires that the key $\boldsymbol{k}$ is not used more than once. Unfortunately, the transmission or synchronization of the key results in less than perfect secrecy. However, this remains an important concept in cryptography.

A block cipher is a bijective function that encrypts a fixed-size block of data with a fixed-size key and outputs a ciphertext with the same size as the corresponding plaintext. Common block sizes are 8 bytes and 16 bytes with a wider range of key sizes. Consequently, it may be required to pad messages until the plaintext size is a multiple of the block size. Students should understand common block ciphers such as `DES`, `3DES`, `AES`, and `Blowfish`. It should be clear that although block ciphers provide confidentiality, they do not provide authenticity, integrity, or non-repudiation. To meet requirements for integrity, it is common to utilize a cryptographic hash function on the ciphertext data and send the hash along with the ciphertext, although the security of this method affected by the cryptographic algorithm that is used. Methods for supporting authenticity and non-repudiation are better provided in asymmetric cryptographic algorithms.

Feistel Networks are block cipher constructions designed to split the block in half, perform complex operations on each half, rejoin the block halves on opposite sides, and repeat. This method has been shown to produce a large variance between plaintext and encrypted blocks, depending on the complexity of the operations performed and the number of repetitions. All block ciphers discussed so far are ultimately substitution ciphers, and both encryption and decryption are deterministic bijective functions. Students should be introduced to different modes of operations used on block ciphers. They should become familiar with Electronic Codebook (ECB), Cipher Block Chaining (CBC), and Counter Mode (CTR). The objective of this section is not to have students implement these cryptographic algorithms, but rather to understand their usage, strengths, and weaknesses. However, it is important for students to understand how to utilize these algorithms in Python. The `pycrypto` library can be used to to encrypt and decrypt data using a variety of block ciphers and modes of operation. If provided enough hints about the block cipher, key, and mode of operation, students should

be able to decrypt ciphertext via brute-force.

The final subject in this section is asymmetric cryptography. Asymmetric cryptography allows two parties to securely communicate without having to agree on shared keys beforehand. Students should understand the concept of public and private keys and how they relate to asymmetric cryptosystems. The essentials of the Diffie-Hellman Key Exchange and RSA algorithm should be explained to students, along with the mathematical concepts behind both methods. Exercises for decrypting RSA encrypted data with relatively small public keys should be given either in class or as part of coursework. Lastly, students should understand the difference between encryption and signing in asymmetric cryptography. Ultimately, a sender encrypts data using the public key of the recipient, and a sender signs data using their private key. In the first case, the receiver can retrieve the original message by decrypting it with their private key. In the second case, the receive can verify the sender of the message by decrypting it with the senders public key. These two methods provide for authorization and non-repudiation, respectively.

Chapter 4

Capture The Flag Events

After covering each section in Chapter 3 students will be ready to compete in cybersecurity Capture The Flag events. At this point in the semester there should two to three remaining weeks. Having a Practice CTF gives students a chance to review the concepts covered throughout the semester before the final. For the Practice CTF, I selected a number of challenges submitted in the Term Project, along with a few of my own, and hosted an event in which students were required to obtain a minimum of 100 possible points. Each challenge offered 15 to 35 points based on its difficulty. Students were allowed to work in groups of two to three people, but they were not allowed to receive credit for any challenges that members of their group helped create. This event was experimental and the development and selection of challenges within the Practice CTF is left to the discretion of future implementors of this work. The final CTF should consist of three to four challenges per category. The categories should include Web, Forensics, Reverse Engineering, Binary Exploitation, and Cryptography. There should minimally be an easy, a medium, and a hard challenge for each category, with each challenge offering a number of points relative to its difficulty. For this event I decided to require a minimum of 100 total points for undergraduate

submissions and 150 total points for graduate submissions. Students were not permitted to work in groups for the Final CTF. Each submission for both the Practice and the Final CTFs took the form of a write-up. At this point in the course, students are expected to be very familiar with how to create and submit write-ups. Submissions should be graded on the basis of challenge completion and write-up quality.

Chapter 5

Conclusions

After implementing this work I found that students enjoyed the course, even though it was very challenging and fast pace. I received positive feedback and appreciation from all students, even those whom struggled the most. It is my hope that the skills acquired throughout the study of this curriculum greatly benefit past, present, and future students. At the time of writing this paper I found that the average assignment score for undergraduate students in Spring of 2019 was roughly 88% while the average assignment score for graduate students in the same course was roughly 93%. I anticipate these grades improving by the time final grades are posted, as the syllabus allows for the two assignments with the lowest grades to be dropped on a per student basis. Additionally, the term project and CTF events are designed for students to perform well, further improving their grades.

To anyone interesting in recreating this work. This document and the appendix items describe in great detail the course objectives and resources utilized throughout the curriculum. You should review the supplemental archive `JacobMills_MSProject.tar` and all included files. Within this archive you will find lectures for each week of the course, homework assignments, binary files, docker configurations, and more.

Appendix A

Term Group Project

Important dates

Proposal and Team Members: 02/28

Submission: 04/04

Number of students in a group

Groups may consist of up to three students. No more than two graduate students are permitted to be in the same group.

Project categories

Projects may be any combination of the following categories. If you would like to include an additional category, please speak with an instructor beforehand.

- Scripting
- Web Exploitation
- Digital Forensics
- Reverse Engineering

- Binary Exploitation
- Cryptography
- Reconnaissance

Objective

For this project you are tasked with the creation of your own CTF challenges. These challenges will be used during the Practice CTF event later in the semester. Therefore, the group you work with on this project will be the same group you work with during the practice CTF. Each challenge that is part of a term project should involve a minimum of one of the topics discussed over the course of this semester. Challenges may vary from easy, medium, and hard difficulties, based on the sophistication of the techniques required for solving them. Your team is required to submit two to three CTF challenges based on their difficulty, respectively. The following guideline will be used to evaluate the difficulty of a challenge:

Easy

This challenge can be solved using simple techniques for string identification, repetitive script interaction, or basic reverse engineering by analyzing control flow mechanisms (either in binary files or web documents).

Medium

This challenge requires a thorough analysis before it can be solved. Solutions may require a combination of one or more techniques from the lectures on digital forensics, web exploitation, reverse engineering, and/or cryptography. Cryptographic, forensic, and/or reversing techniques may be required to solve this challenge; for

example: a custom algorithm might be used to scramble the data of a target string or file. Remote shells may be acquired by simple techniques such as a buffer overflow. A combination of advanced techniques was not used during the creation of this problem.

Hard

This challenge requires a thorough analysis before a solution can be attempted. Solutions may require a combination of one or more techniques from the advanced lectures on digital forensics, web exploitation, reverse engineering, and/or cryptography. This problem might use one or more advanced techniques in Cryptography such as AES encryption or the discrete log problem. Dynamic analysis may be required as part of the solution. Remote shells (pwn) are only attainable by exploiting a stack canary or higher-level binary security construct. Remote code execution (web) is only possible after a security vulnerability for escalating privileges is first exploited. A combination of advanced techniques is required to solve this problem. Problems must be at a similar level of difficulty as that of the advanced topic assignments.

Proposal

The project proposal should include team members (up to three students per team - only two graduate students per team) and an outline of the challenges that your team will attempt to create. A technical description of the challenges is not required. Only the categories, required background and utilities, summaries, and estimated difficulty of problems to be created should be submitted as part of the proposal. Proposals should be submitted during class on February 28th to ensure that all students have a group.

Grading

This is an optional assignment for undergraduate students; however, undergraduates can earn up to 10% extra credit toward their final grade by participating in the term project. This is a mandatory project for graduate students will be a 10% of their final grade. Your group is required to submit challenges with one the following difficulty combinations:

- Easy, Medium, Hard
- Medium, Medium, Medium
- Hard, Hard

Your group will be evaluated on the novelty of your created challenges, their accessibility (can we actually access the challenges and attempt to solve them?), their association with one or more of the permissible categories, and their respective difficulties. Note: for challenges that require network interaction, **you will be responsible for establishing the hosting environment. Please be EXTREMELY CAREFUL when doing so, as we will not be liable for any damages or harm that may result from creating a vulnerable computing environment accessible over a network.** For advice on how you can create network-based challenges in a safe environment please speak with the course instructors. The project will be scored out of 100 possible points based on the following criteria:

Accessibility of challenges	5 pts
Quality of README file	5 pts
Proposal	10 pts
Appropriate difficulty of challenges	15 pts
Novelty of created problems	15 pts
Quality of Assignment Creation Document	25 pts
Completeness of challenges	25 pts

Submission

You are required to fill out the Challenge Creation Template for each challenge that is a part of your submission (see course homepage for link). A single member of each group should submit a tar file containing each completed challenge creation template, all source files, and a simple README document listing group members, their respective grade levels (graduate or undergraduate), and an explanation of the contents of the archive.

Practice CTF

The Practice CTF will include instructor designed challenges and selected student designed challenges from submitted term projects. This will be a group exercise over the course of a week, and groups will consist of the same team members as the term project. Undergraduate students without a term project group will be optionally assigned to a group during the week of the Practice CTF. The Practice CTF is required for all students and will be 10% of your final grade. Groups in the practice CTF will receive zero points for completed challenges that were designed by their group. In other words, you will only receive points for the challenges that no member of your group helped create. A single submission in the form of

a collective assignment submission template is required for each group.

Appendix B

Assignment Creation Template

Introduction

Title: A title for this challenge

Category: The primary challenge category (ie: Web, Forensics, ...)

Themes: Sub-themes of the challenge (ie: SQL-injection, file recovery, ...)

Difficulty: With respect to this category, what is the intended difficulty of this problem?

Objectives: Areas in which students will improve upon by completing this challenge. Example:

- Develop skills with Python scripting for Web interaction
- Gain experience with URL encodings

Overview

Explanation

Brief overview of the challenge. The challenge as it is described to students. Example:

```
Using Python, decode this URL encoded string:
%65%78%61%6d%70%6c%65
```

Prerequisite Knowledge

- Python scripting
- Python lists
- Hashing functions

Details

Implementation

Detailed description of the challenge. The challenge as it is described to instructors. Example:

This problem offers a network service listening on an IP address and port provided to students. After a connection is established, the server will write a hashed value to the client. The value comes from the result of performing a MD5 hash on a random word from the English dictionary. After the student successfully provides input that results in an identical hash, a new hashed value is provided. The student must successfully provide input for three iterations before the server provides the flag.

Methodology

Primary steps to solving this challenge. For example:
Step 1: load wordlist into an array
Step 2: connect to provided server IP address and port
Step 3: perform a hash of each wordlist item
...

Suggested Rubric

An example rubric is provided below:

Quality of write-up	5 pts
Clarification of the task to be completed	5 pts
Successful loading of dictionary into a list	5 pts
Connecting to server and extracting the target value	5 pts
Brute force solving the problem	5 pts
Successfully identifying the flag	5 pts

Notes

Ways in which this problem could be improved, grading lenience, additional information

Appendix C

Assignment Submission Template

Overview

Assignment: Which assignment is this?

Student: Who are you?

Date: What is the date?

Problem 1

Problem: Which problem is this?

Analysis: What is the goal of this challenge? If any vulnerabilities exist, discuss them and how they might be exploited.

Plan: Before solving this problem, discuss techniques that should be considered for finding a solution.

Solution: Discuss step-by-step the actions you performed to solve this challenge. Feel free to include screenshots, code, or anything else that was useful during your analysis.

Flag: What is the flag?

Problem X

Problem: Which problem is this?

Analysis: What is the goal of this challenge? If any vulnerabilities exist, discuss them and how they might be exploited.

Plan: Before solving this problem, discuss techniques that should be considered for finding a solution.

Solution: Discuss step-by-step the actions you performed to solve this challenge. Feel free to include screenshots, code, or anything else that was useful during your analysis.

Flag: What is the flag?

Optional Feedback

Do you feel like previous lectures adequately prepared you for this assignment?

How difficult did you find this assignment?

Do you feel like this assignment helped develop your skills in the associated subject?

Is there anything about the assignment or lecture that you enjoyed or thought could be improved?

Appendix D

CTFd docker-compose.yml

```
version: '2'

services:
  ctfd:
    build: .
    restart: always
    ports:
      - "8000:8000"
    environment:
      - UPLOAD_FOLDER=/var/uploads
      - DATABASE_URL=mysql+pymysql://root:ctfd@db/ctfd
      - REDIS_URL=redis://cache:6379
      - WORKERS=4
    volumes:
      - .data/CTFd/logs:/var/log/CTFd
      - .data/CTFd/uploads:/var/uploads
      - .:/opt/CTFd:ro
    depends_on:
      - db
    networks:
        default:
        internal:

  db:
    image: mariadb:10.4
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=ctfd
```

```
      - MYSQL_USER=ctfd
      - MYSQL_PASSWORD=ctfd
    volumes:
      - .data/mysql:/var/lib/mysql
    networks:
        internal:
    # This command is required to set important mariadb defaults
    command: [mysqld, --character-set-server=utf8mb4, --collation-server=utf8mb4

  cache:
    image: redis:4
    restart: always
    volumes:
    - .data/redis:/data
    networks:
        internal:

networks:
    default:
    internal:
        internal: true
```

Appendix E

/etc/nginx/sites-enabled/CTFd

```
server {
    server_name fsuctf.com;
    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
      root /home/CTFd/CTFd/CTFd;
    }
    location / {
        proxy_pass http://fsuctf.com:8000;
        index  index.html index.htm;
    }
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/fsuctf.com/fullchain.pem; # managed by
    ssl_certificate_key /etc/letsencrypt/live/fsuctf.com/privkey.pem; # managed
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
}

server {
    if ($host = fsuctf.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot
  listen 80;
  server_name fsuctf.com;
    return 404; # managed by Certbot
}
```

Appendix F

Sample Python Dockerfile

```
FROM ubuntu:trusty

RUN apt-get update
RUN apt-get update && apt-get -y dist-upgrade
RUN apt-get install -y ucspi-tcp
RUN apt-get install -y python

RUN useradd -m ctf
COPY * /
RUN chmod +x /challenge.py

WORKDIR /

EXPOSE 31337
USER ctf
CMD tcpserver -t 50 -RHl0 0.0.0.0 31337 ./challenge.py
```

Appendix G

LEMP docker-compose.yml

```
nginx:
    image: tutum/nginx
    ports:
        - "8080:8080"
    links:
        - phpfpm
    volumes:
        - ./nginx/default:/etc/nginx/sites-available/default
        - ./nginx/default:/etc/nginx/sites-enabled/default
        - ./logs/nginx-error.log:/var/log/nginx/error.log
        - ./logs/nginx-access.log:/var/log/nginx/access.log
        - ./docker-static/img:/usr/share/nginx/html/img/
phpfpm:
    build: .
#    image: php:fpm
    ports:
        - "9001:9001"
    volumes:
        - ./public:/usr/share/nginx/html
        - ./php.ini:/usr/local/etc/php/conf.d/custom.ini
    links:
        - mysql
mysql:
  image: mysql
  environment:
    MYSQL_ROOT_PASSWORD: omitted
    MYSQL_USER: root
    MYSQL_PASSWORD: omitted
```

```
    MYSQL_DATABASE: dockerdb
  ports:
    - "3306"
phpmyadmin:
  image: phpmyadmin/phpmyadmin
  restart: always
  links:
    - mysql
  ports:
    - 8183:80 # should be 8183:80
  environment:
    PMA_HOST: mysql
```

Appendix H

Sample xinetd Configuration

```
service ctf
{
    disable = no
    socket_type = stream
    protocol    = tcp
    wait        = no
    user        = root
    type        = UNLISTED
    port        = 9999
    bind        = 0.0.0.0
    server      = /usr/sbin/chroot
    server_args = --userspec=1000:1000 /home/ctf/ ./init.sh
    banner_fail = /etc/banner_fail
    # safety options
    per_source = 10 # the maximum instances of this service
    rlimit_cpu = 20 # the maximum number of CPU seconds
    #rlimit_as  = 1024M # the Address Space resource limit
    #access_times = 2:00-9:00 12:00-24:00
}
```

References

[1] G. C. Kessler and J. Ramsay, "Botnets, cybercrime, and cyberterrorism: Vulnerabilities and policy issues for congress," tech. rep., Journal of Homeland Security Education, January 2013. ii

[2] G. C. Kessler and J. D. Ramsay, "A proposed curriculum in cybersecurity education targeting homeland security students," in *2014 47th Hawaii International Conference on System Sciences*, pp. 4932–4937, Jan 2014. ii

[3] D. Inc., "What is a container?," 2019. 11

[4] K. Angrishi, "Turning Inernet of Things (IoT) into Internet of Vulnerabilities (IoV) : IoT botnets," *CoRR*, vol. abs/1702.03681, 2017.

[5] J. Kallberg and B. Thuraisingham, "Towards cyber operations - the new role of academic cyber security research and education," in *2012 IEEE International Conference on Intelligence and Security Informatics*, pp. 132–134, June 2012.

[6] G. C. Kessler and J. Ramsay, "Paradigms for cybersecurity education in a homeland security program," *Journal of Homeland Security Education*, vol. 2, 2013.

[7] D. H. Tobey, P. Pusey, and D. L. Burley, "Engaging learners in cybersecurity

careers: Lessons from the launch of the national cyber league," *ACM Inroads*, vol. 5, pp. 53–56, Mar. 2014.

[8] C. Paulsen, E. McDuffie, W. Newhouse, and P. Toth, "Nice: Creating a cybersecurity workforce and aware public," *IEEE Security Privacy*, vol. 10, pp. 76–79, May 2012.

[9] L. D. Paulson, "Developers shift to dynamic programming languages," *Computer*, vol. 40, pp. 12–15, Feb 2007.

[10] M. Sikorski and A. Honig, *Practical Malware Analysis*. William Pollock, 6 ed., 2012.

[11] R. O'Neill, *Learning Linux Binary Analysis*. Packt Publishing Ltd., 1 ed., 2016.

[12] J. Erickson, *Hacking: The Art of Exploitation*. William Pollock, 2 ed., 2008.

[13] M. Stamp, *Information Security*. John Wiley & Sons, Inc., 2 ed., 2011.

[14] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook*. John Wiley & Sons, Inc., 2 ed., 2011.