8-19-2019

# BootBandit: A macOS bootloader attack

Armen Boursalian
*San Jose State University*

Mark Stamp
*San Jose State University*, mark.stamp@sjsu.edu

Received: 27 May 2019 | Revised: 25 July 2019 | Accepted: 28 July 2019

DOI: 10.1002/eng2.12032

RESEARCH ARTICLE

WILEY

# BootBandit: A macOS bootloader attack

**Armen Boursalian** | **Mark Stamp**

Department of Computer Science, San Jose State University, San Jose, California

**Correspondence**
Mark Stamp, Department of Computer Science, San Jose State University, San Jose, CA 95192-0249.
Email: mark.stamp@sjsu.edu

Historically, the boot phase on personal computers left systems in a relatively vulnerable state. Because traditional antivirus software runs within the operating system, the boot environment is difficult to protect from malware. Examples of attacks against bootloaders include so-called "evil maid" attacks, in which an intruder physically obtains a boot disk to install malicious software for obtaining the password used to encrypt a disk. The password then must be stored and retrieved again through physical access. In this paper, we discuss an attack that borrows concepts from the evil maid. We assume exploitation can be used to infect a bootloader on a system running macOS remotely to install code to steal the user's password. We explore the ability to create a communication channel between the bootloader and the operating system to remotely steal the password for a disk protected by FileVault 2. On a macOS system, this attack has additional implications due to "password forwarding" technology, in which a user's account password also serves as the FileVault password, enabling an additional attack surface through privilege escalation.

**KEYWORDS**
bootloader, evil maid attack, Mac OS

## 1 | INTRODUCTION

The "evil maid" attack gets its name from a hypothetical situation in which, say, 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 an encrypted computer system. The next time the computer is used, the malware steals the encryption password.

Such an attack takes advantage of the vulnerable state of a computer system before it boots into its operating system environment. In this preboot 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. The goal of an evil maid attack is to obtain a full disk encryption (FDE) password to be able to decrypt a disk drive. This generally assumes that 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 is, in most systems, used only for 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 preoperating system environment also means theft of the login credentials, which, on a personal computer, is often also sufficient for gaining root or 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.

We note that the primary goal of this research is to abuse the bootloader and Apple's "password forwarding" technology, as demonstrated by stealing a user's credentials. However, unlike a traditional evil maid attack, our attack is not intended to be a physical one. Our goal is to demonstrate the possibility of using the UEFI space to communicate an attack to user space. Because user passwords on macOS systems typically double as disk encryption passwords (and triple as administrator passwords), theft of the password was the ideal target to showcase such an attack.

In the attack chain, we assume that prior exploitation allows filesystem access to the protected `/System` directory but not root privileges. Vulnerabilities like this have been discovered in the past.[1] Such exploits do not always provide indefinite or sustained root access, so an attack like BootBandit, although obviously not the only option, can be used to gain credentials to escalate privileges and continue lateral movement. Moreover, again, we demonstrate use of UEFI to bypass any runtime defenses and to communicate from the boot level to the OS level using UEFI facilities.

The attack discussed in this paper was tested on OS X 10.11.6. Our attack applies generically to relevant hardware that is not protected by the T2 chip.[2]

The document[2] states that

> For Mac computers with the AppleT2Security Chip, each step of the startup process contains components that are cryptographically signed by Apple to verify integrity … . The boot process proceeds only after verifying the integrity of the software at every step, which creates a chain of trust rooted in hardware. This includes the UEFI firmware, bootloaders, kernel, and kernel extensions necessary for boot. This secure boot chain helps ensure that the lowest-level software is not tampered with, so the Mac computer will be in a known trustworthy state when it is booted.

Hence, a modified bootloader attack such as BootBandit will not succeed against hardware that is protected by the T2 security chip. On the other hand, for hardware that does not utilize the T2 chip for bootloader protection, the precise version of the OS does not matter, provided the attacker can gain filesystem access to the `/System` directory, as mentioned above. Note that the T2 chip was released in 2018.

In Section 2, we introduce the concepts of disk encryption and the PC boot process, we discuss how these apply specifically to macOS, and we outline selected previous work involving attacks against systems in the boot phase and attacks on FDE systems. Section 3 discusses the implementation of our BootBandit system, which is a bootkit that harvests user credentials that are collected by a user mode implant and then exfiltrated to our command and control server. Details on the development of the attack are discussed in Section 4, while potential defenses against such an attack are outlined in Section 5. Finally, in Section 6, we conclude and provide a discussion of future work that can build upon BootBandit.

## 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 preboot environment. We also discuss some of the key components of the boot process for macOS that will be the 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 purpose of FDE is to maintain confidentiality when an adversary gains physical access to a device. 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 true regardless of 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. To render such stolen disk drives useless to thieves, FDE can be 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 and 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 the

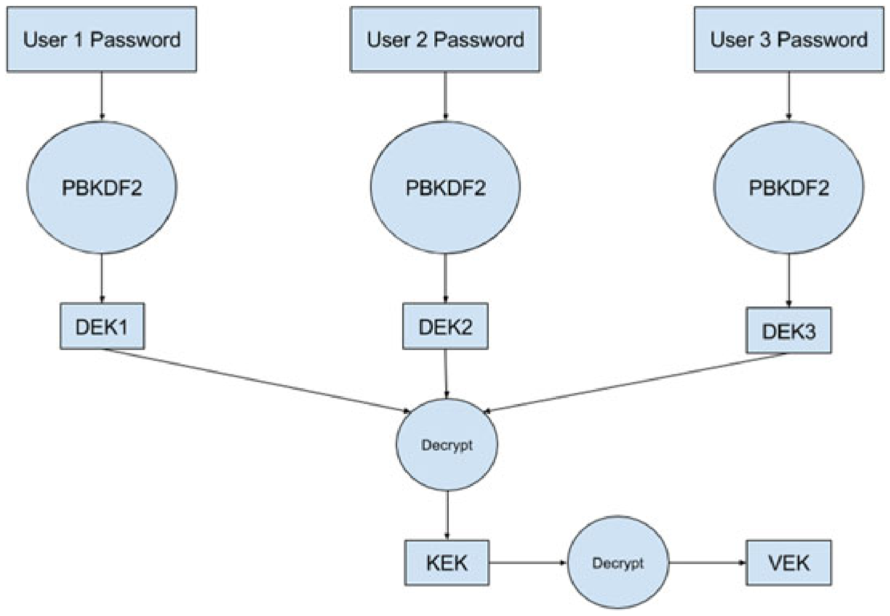


**FIGURE 1** FileVault 2 key management. DEK, derived encryption key; KEK, key encryption key; PBKDF2, password-based key derivation function 2; VEK, volume encryption key

user password. Note that if there are multiple users of a system, only one derived encryption key (DEK) is necessary so that users can unlock the system individually.[3]

### 2.1.1 | macOS

Since Mac OS X 10.7.0 (before the operating system was rebranded as "macOS"), FileVault 2 has been Apple's default technology for disk encryption. It is an improvement over legacy FileVault in many ways. For example, FileVault only encrypted individual users' account directories but not the entire disk, whereas FileVault 2 is an actual full disk encryption technology. 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 three-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 explained in detail below.

At the volume level, AES-XTS-128 is used to encrypt data blocks on the disk.[4] Despite the name, this encryption scheme uses a 256-bit key.[5] It is designed specifically for encrypting stored data as opposed to data in transit, eg, data sent over a network. Apple calls the master AES-XTS-128 key the *volume encryption key*, or VEK.

To obtain the decrypted VEK, a KEK is used as discussed above. 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* (PBKDF2) algorithm is applied to the passphrase to obtain the DEK. This three-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 other's and without requiring the entire volume to be re-encrypted with a new key. It is the passphrases belonging to the individual users that are the subject of our attack.

## 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 (EFI). The EFI standard was developed by Intel and released in 1999 for its Itanium® processor architecture. The industry needed a successor for legacy BIOS, and hence, in 2005, the UEFI specification was published. Moreover, 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.

From the end user's perspective, the most readily apparent benefit of UEFI over legacy BIOS 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 in UEFI is the bootloader.

### 2.2.1 | UEFI features

Here, UEFI provides many facilities for operating systems to interface with the underlying firmware. These features are provided as "services" that are split between *boot services* and *runtime services*.[6] 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 (eg, to install firmware updates from the operating system environment), and key/value data storage in firmware RAM. This latter point is the focus of this project.

Key/value data can be stored in the firmware RAM,[7] both for use in the boot phase and by the operating system. In macOS, for example, the audio volume level is stored in nonvolatile RAM (NVRAM), 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. In BootBandit, we use this resource 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 that runs in a preoperating 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 a requirement of the UEFI specification and has no relationship to the target operating system that is to be loaded.

Like all other UEFI applications, the entrypoint for execution is the function `UefiMain`, the signature of which is shown in Listing 1. This is analogous to the `main` function for C programs that run on most operating systems.

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 the aforementioned boot services and runtime services. This parameter will play a key role in the BootBandit attack, as we use it 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 can be loaded. Any attack mitigations must be in place in the firmware, which then passes execution to the bootloader. Moreover, because the bootloader is responsible for loading the operating system, the implications of bootloader attacks can range from password theft of FDE devices to backdoors that are virtually undetectable once in user mode.

Prior work in the bootloader attack space goes back to the 1980s. At that time, there was no UEFI, and booting was done with whatever code was in the hard disk drive's MBR. It was easy to infect the code in the MBR, whose purpose was to read the filesystem on a disk and boot the operating system. The Brain virus was released into the wild in 1986 and is often considered the first computer virus for MS-DOS systems.[8] For its success, the Brain virus relied on infecting the MBRs of boot disks.

Previously, attacks similar to BootBandit targeting other FDE technologies have succeeded. In 2009, renowned security researcher Joanna Rutkowska published a proof-of-concept of an "evil maid" attack targeting the TrueCrypt FDE system.[9] 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 send the password over a network, since physical access would be required again to decrypt the disk unless it was copied during installation

**Listing 1** Unified extensible firmware interface application entrypoint

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

of the malware. Therefore, the attack as given in the work of Rutkowska[9] did not include network capabilities. BootBandit builds on a traditional "Evil Maid" attack in these areas.

More recently in 2015, security researcher Pedro Vilaca spoke at the Code Blue conference,[10] detailing 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 antivirus to detect. Although no proof-of-concept was developed for this talk, tools targeting UEFI written by the Italian company "Hacking Team" were mentioned, along with concepts such as nearly invisible malware and disk encryption theft. The work by Vilaca[10] served as the primary inspiration for BootBandit.

Additional recent work related to bootloader attacks include the works of Guan et al[11] and Redini et al,[12] whereas attacks on FDE systems are discussed in other works.[13-15] Both bootloader and FDE attacks in the context of IoT security are considered in the work of Johnston et al.[16]

Redini et al[12] have developed a taint analysis system, BootStomp, which is designed to find potential security flaws in bootloaders. Using their BootStomp system, the authors were able to discover six previously unreported security flaws in four different systems.

The work of Bossi and Visconti[13] is focused on the Linux Unified Key Setup (LUKS) full disk encryption specification. The authors show that users can unwittingly create security vulnerabilities in a LUKS system by choosing certain hashing options or by choosing aggressive power management options.

The research in the work of Johnston et al[16] is focused on IoT security. Among other things, the authors recommend bootloader signing and bootloader encryption as ways to mitigate various attacks.

In the work of Khati et al,[14] full disk encryption is considered under the assumption that neither IVs nor MACs are used. In typical cryptographic scenarios, MACs and IVs generally serve to provide a cryptographical integrity check and a significant "indistinguishability" feature,* respectively. The authors develop a theoretical framework and show how to deal with these security issues to some extent under the specified constraints.

Guan et al[11] present a cryptographic engine, Copker, which they have developed. Copker performs cryptographic operations within the CPU so that no "plaintext-sensitive" data is stored in RAM. This is relevant in the context of FDE, since cryptographic information in RAM can be vulnerable to capture.

The research in the work of Müller and Freiling[15] is focused on a comparison of software-based and hardware-based FDE. The authors demonstrate attacks on FDE standards of both types and conclude that neither approach is inherently superior to the other.

## 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 NVRAM that the machine is to boot an operating system using this particular file. Of course, 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.[4] On older Apple computers, the kernel is located at the 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 Apple's "password forwarding" technology[4] makes the attack possible because it allows the user's password to both unlock the disk and log into the account.

*In this usage, indistinguishability means that identical plaintext does not encrypt to identical ciphertext. While this might seem like an esoteric academic concern, it is actually a surprisingly important issue.[3]

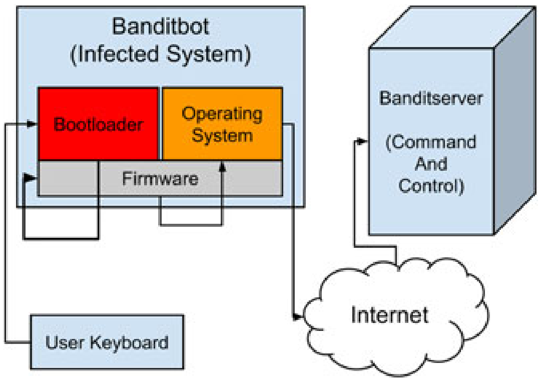


**FIGURE 2** BootBandit architecture

## 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 (from the attacker's perspective) of revealing a user's login credentials. In effect, "password forwarding" 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 three main components:

- Modified bootloader (BootBandit);
- User mode implant (BanditBot);
- Exfiltration server (BanditServer).

The architecture for the attack is outlined in Figure 2. This figure illustrates 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 that is responsible for stealing a user's login credentials. It has been developed so that the user experience during login is unaffected and the theft occurs transparently in the background. The modification is an infection of the official bootloader, where the credential-stealing code has been placed in a "code cave," ie, 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 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 can carry 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 writes the user's credentials 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. 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[17] 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 Apple did not compile the bootloader to support a UEFI debugger. 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 invariably led to a crash.

```
    "_LW_LoginPane_ValidatePassword"
    "VerifyCallback␣is␣NULL"
    "loginUI->loginUICallbacks.VerifyPassphraseFunction␣is
      ␣NULL"
    "lw->verifyPasswordFunction␣is␣NULL"
```

**Listing 2** Plaintext strings referring to password functionality

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 2, which pertain to password validation and verification.

The strings in Listing 2 are meant for the developers of the bootloader to be able to debug the program. However, they also provide us, acting as the attacker, with important context as to how we can insert malicious code. Specifically, the string

`"_LW_LoginPane_ValidatePassword"`

is passed as a function name to an error message 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, in which case the bootloader would otherwise have run into a segmentation fault. This tells an attacker exactly at which offset the password verification function is located, and the object can then be traced back through cross-references to find which function is responsible for verifying the user's password input.

We begin with the function `UnlockCoreStorageVolumeKey`, as identified by the debugging string in the main function. Everything prior to this call is 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 eight parameters, the last of which is a pointer to a function referencing the struct `PassphraseWrappedKekStruct` in a dictionary lookup. We know from Section 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 was performed in IDA Pro on the disassembly. We find 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 and it is 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 is illustrated in Figure 3.

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 `VerifyCallbackfunction` takes the user's password as a parameter, finds its length, and uses it to calculate

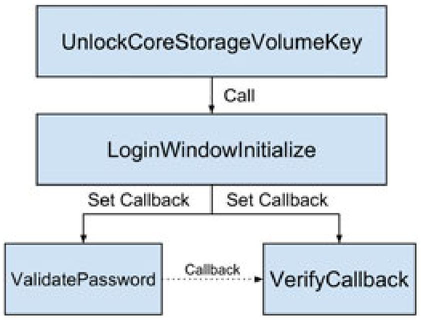


**FIGURE 3** Password verifier callback setup

the DEK. Now knowing the precise location where the user's password will be passed, we can infect the bootloader to obtain the password.

### 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," `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 the original (hooked) function that was replaced; otherwise, login cannot proceed, and the user will have a nonbooting 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 `nvram` 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 is the function call, translated to C, which 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 the `SetVariable` 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 it appears that it is only possible to write to the table pointer from the following three places in the code:

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

The first of these three 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` has pointers to both the `BootServices` and `RuntimeServices` tables.[7] Ultimately, the pointer indirection shown in Figure 4 was used to store the user's credentials in the firmware memory. The full disassembly of the infection code can found in the appendix of the report.[18]

## 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` 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 can make use of it. This firmware variable is used as a communication channel between the modified bootloader in the preoperating system environment and the backdoor in the desktop environment, where network access is easy.

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

**Listing 3** Writing credentials to firmware memory

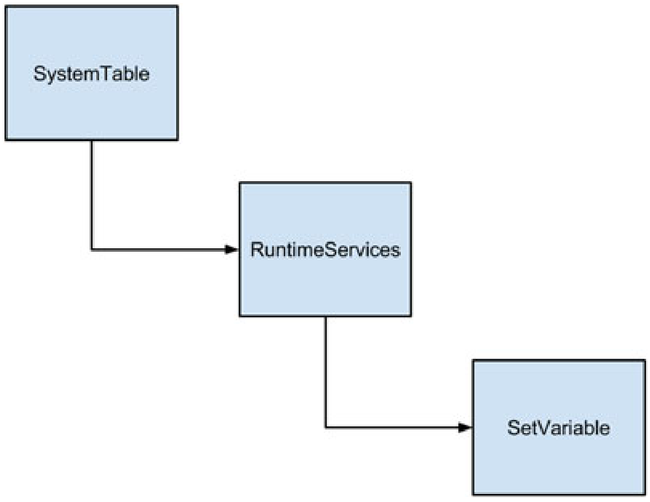


**FIGURE 4** Obtaining runtime services to call `SetVariable()`

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 `nvram`, which allows reading and writing of firmware variables, both volatile and nonvolatile, 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[19] 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 strong network communication. At this point, exfiltration is complete.

## 3.3 | Exfiltration server

The BootBandit exfiltration server is written in Go[20] for rapid development and portability. It runs in a Google Compute Engine virtual machine on the Google Cloud Platform and is accessible worldwide, serving here as a proof-of-concept where infected systems can send exfiltrated information back to the attacker. The BootBandit network protocol is illustrated 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, four-byte, little-endian integer.

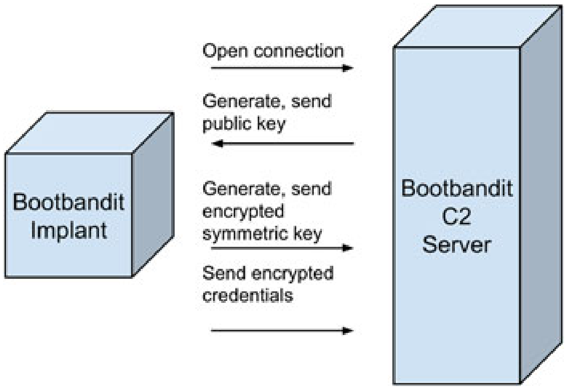

Open connection
Generate, send public key
Generate, send encrypted symmetric key
Send encrypted credentials
Bootbandit Implant
Bootbandit C2 Server

**FIGURE 5** BootBandit command and control protocol

The credential data is stored in a log file which the attacker can use to collect IP addresses, user names, and passwords of victims.

## 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 injecting 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. However, IDA Pro, 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

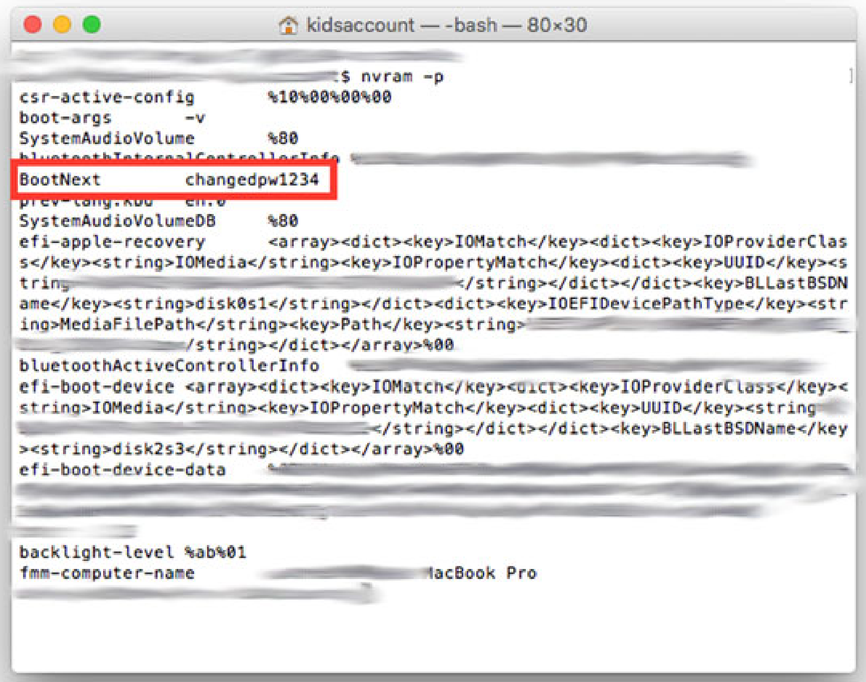
After discovering that IDA Pro failed to load the disassembly at the virtual addresses declared in the PE header, as described in Section 4.1.1, the appropriate corrections were made to the bootloader infection code using `Radare2`. However, the problem seemed to remain unresolved. This was due to the issue discussed in Section 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 with the `SetVariable` function, a modification to the bootloader was made so that the memory address of the data structure we were looking for, namely `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. Moreover, 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 the function `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` 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 include 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 Section 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 is depicted in Figure 4. After using the `RuntimeServices` structure indirectly, we were able to successfully obtain the user's password and place it in the `BootNext` variable in the firmware. This is shown in Figure 6.



```
kidsaccount — -bash — 80×30
[redacted]$ nvram -p
csr-active-config       %10%00%00%00
boot-args         -v
SystemAudioVolume       %80
bluetoothInternalControllerInfo [redacted]
BootNext         changedpw1234
prev-lang:kbd     en:0
SystemAudioVolumeDB     %80
efi-apple-recovery      <array><dict><key>IOMatch</key><dict><key>IOProviderClas
s</key><string>IOMedia</string><key>IOPropertyMatch</key><dict><key>UUID</key><s
trin[redacted]</string></dict></dict><key>BLLastBSDN
ame</key><string>disk0s1</string></dict><dict><key>IOEFIDevicePathType</key><str
ing>MediaFilePath</string><key>Path</key><string>[redacted]
[redacted]/string></dict></array>%00
bluetoothActiveControllerInfo [redacted]
efi-boot-device <array><dict><key>IOMatch</key><dict><key>IOProviderClass</key><
string>IOMedia</string><key>IOPropertyMatch</key><dict><key>UUID</key><string[redacted]
[redacted]</string></dict></dict><key>BLLastBSDName</key
><string>disk2s3</string></dict></array>%00
efi-boot-device-data    [redacted]

backlight-level %ab%01
fmm-computer-name       [redacted]acBook Pro
[redacted]
```

**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 could be used, but this path is readily accessible and hidden from the user. The implant is set to run on system start and send the credentials to the command and control server by creating the file

`/Library/LaunchAgent/com.user.persist.plist`.

This file has the contents shown in Listing 4.

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

```
<?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** File `.plist` for implant persistence

```
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 5** User credential log

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. Moreover, if the implant itself were to be found and analyzed, the location of the server would not be found unless the configuration file was 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 | 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 5 that the user's password has been stolen, all without any change in the user experience.

The entire attack chain worked for all users on the system, with one exception. The account `kidsaccount`, as seen in Listing 5, 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 situation by placing the implant executable at

```
/Applications/.mal
```

and placing the persistence mechanism at

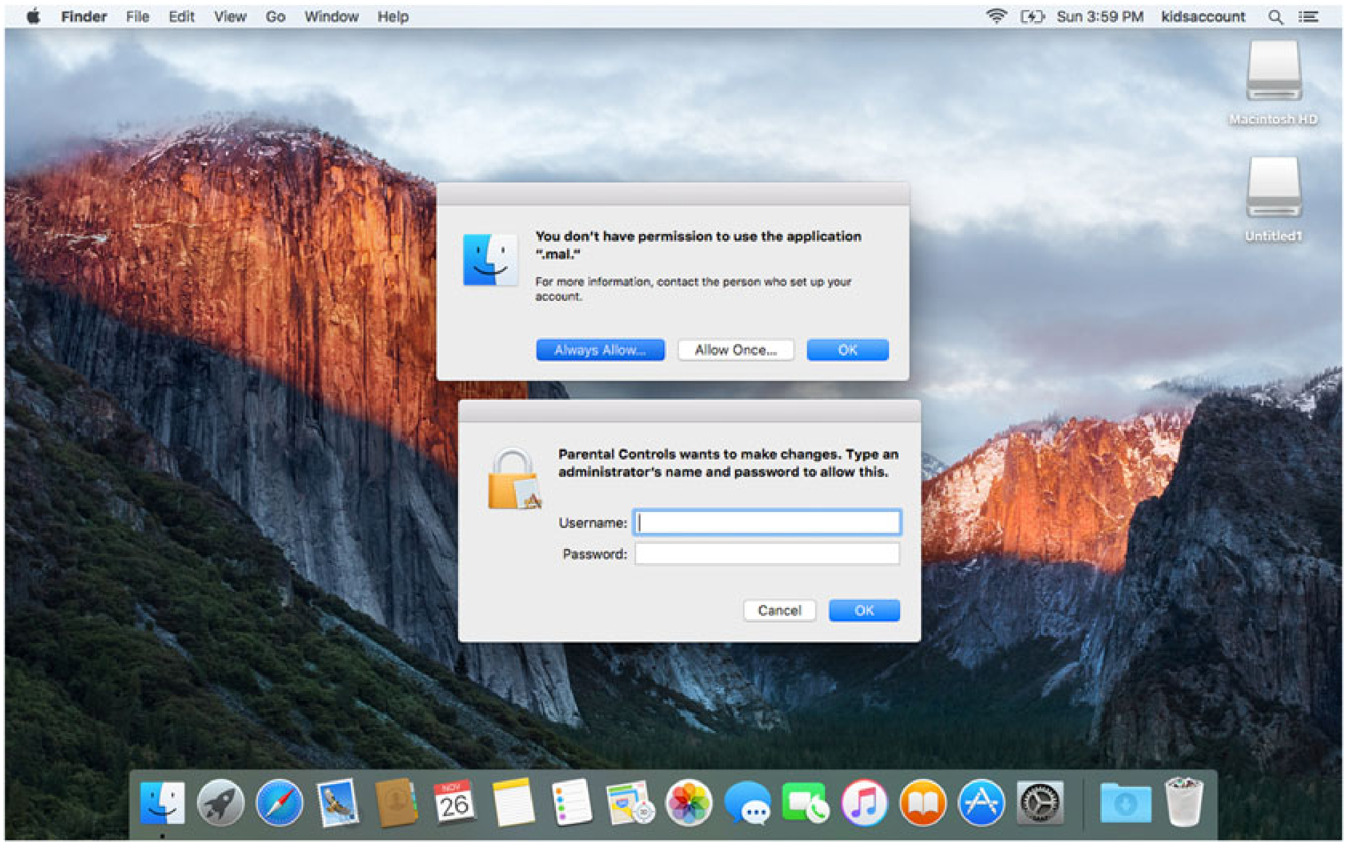
```
/Library/LaunchAgents
```

so that it is shared among all users. The result was the same; accounts with parental controls remained unable to run the implant. Once a system is infected with BootBandit, the parental controls appear to be the only readily available defense in macOS.

Another small issue that was observed was in some 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 three 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.

## 5 | MITIGATIONS AND DEFENSES

In macOS systems, system integrity protection[21] is 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


You don't have permission to use the application ".mal."

For more information, contact the person who set up your account.

Parental Controls wants to make changes. Type an administrator's name and password to allow this.

Username:

Password:

**FIGURE 7** Implant blocked by parental controls

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, in which the boot and system partitions are verified for integrity using code-signing methods.[22] Apple iPhones running iOS[23] and some systems running Windows 8.1[8,24] and above implement similar countermeasures against bootkits. The disadvantages of this sort of defense include making it more difficult to use custom software, for example, to use a different operating system than the one that shipped with the system.

In addition, Apple could make 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 that 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 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 obvious names, the time required to develop 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 discouraged.[3] This is because a password obtained for one account provides an attacker access to all accounts that use this same password. The situation is similar for FDE and user accounts; if an attacker obtains credentials for FDE, then the user's system account is also compromised. For a more secure system, it is advisable to separate the FDE password from a user's credentials. However, an obvious disadvantage of separating these is that users must now remember two passwords. In this case, Apple chose convenience for users over security.

## 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 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 research paves the way for several possible lines of future work. From the perspective of the attacker, we would want to better protect the bootloader, as it is subject to signature scanning or hash-based detection. Standard techniques to evade signature detection include polymorphic and metamorphic code, while hash-based detection can be often be foiled by attacking the hash storage scheme, for example.[25] These approaches would seem to be as applicable in the modified bootloader case considered in this paper as in the general malware scenario.

More fundamental would be the development of an extension of BootBandit to 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. For such an approach to succeed, we 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. With all of the malicious logic of BootBandit implemented in a UEFI driver hidden from disk, the resulting malware would be extremely stealthy.

Developing a defensive mechanism such as the verified boot framework that was described in Section 5 would be an interesting line of research. For this to work, we could develop a UEFI driver using the secure boot and driver signing features as described in the UEFI specification.[6] This driver would be responsible for verifying the bootloader against its code signature. The developer would generate a key pair for signing, then sign the bootloader, and the framework would ensure that only a bootloader that is signed with the private key is allowed to execute. Unfortunately, this approach would also have the negative side effect of forbidding third party UEFI applications from executing on the system; nevertheless, it would serve as a proof-of-concept for a verified bootloader. A productized version of this technique would require first party support from Apple.

The examples above involve the development of a firmware driver, that is, code that controls the underlying hardware. A bug in a UEFI driver can cause damage from which recovery is extremely difficult. Consequently, before beginning work such as that suggested here, it would be necessary to be intimately familiar with reflashing. In some cases, hardware memory devices used for firmware storage might even need to be replaced.

Another direction for future work would be to flesh out the payload beyond password stealing so that we could control the boot process. If this could be accomplished, we would be in position to implement a wide variety of attacks, including the following.

- Modify kernel extensions or drivers.
- Hide malicious processes by modifying kernel code before it is loaded, thus allowing for more advanced spying activity after the user is logged in to the OS. This could include remote access via the command and control unit, ie, the BootBandit C2 server that is illustrated in Figure 5.
- Execute advanced malicious code in boot mode.
- Possibly, set a firmware password and lock the computer from the user. If successful, this could be used as part of a ransomware scheme.

Finally, it would be interesting to determine the viability of BootBandit in the face of recent security developments from Apple. Specifically, the Startup Security Utility[26] allows for a Secureboot[27] mode, features that are only available on Mac computers that have the Apple T2 Security Chip.[28] It appears that Secureboot mode implements concepts that are analogous to the signed bootloader mentioned above. The research that forms the basis of this paper was completed before these security developments from Apple were available.

## ORCID

*Mark Stamp* https://orcid.org/0000-0002-3803-8368

## REFERENCES

1. The Hacker News. Here's the exploit to bypass Apple security feature that fits in a Tweet. 2016. https://thehackernews.com/2016/03/sip-exploit-code.html
2. Apple Inc. Apple T2 security chip: Security overview. 2018. https://www.apple.com/mac/docs/Apple_T2_Security_Chip_Overview.pdf
3. Stamp M. *Information Security: Principles and Practice*. 2nd ed. Hoboken, NJ: John Wiley & Sons, Inc; 2011.
4. Apple Inc. Best practices for deploying FileVault 2. 2012. https://web.archive.org/web/20170829134944/http://training.apple.com/pdf/WP_FileVault2.pdf
5. Keller SS, Hall TA. The XTS-AES validation system (XTSVS). National Institute of Standards and Technology (NIST). 2013. https://csrc.nist.gov/csrc/media/projects/cryptographic-algorithm-validation-program/documents/aes/xtsvs.pdf
6. Unified EFI Forum. Unified extensible firmware interface specification. 2017. http://www.uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf
7. Zimmer V, Rothman M, Marisetty S. *Beyond BIOS: Developing With the Unified Extensible Firmware Interface*. Berlin, Germany: Walter de Gruyter GmbH; 2010.
8. Matrosov A, Rodionov E, Bratus S. *Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats*. San Francisco, CA: No Starch Press, Inc; 2018.
9. Rutkowska J. Evil Maid goes after TrueCrypt! 2009. https://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html
10. Vilaca P. Is there an EFI monster inside your Apple? Paper presented at: CODE BLUE 2015; 2015; Tokyo, Japan.
11. Guan L, Lin J, Ma Z, Luo B, Xia L, Jing J. Copker: a cryptographic engine against cold-boot attacks. *IEEE Trans Dependable Secure Comput*. 2018;15(5):742-754.
12. Redini N, Machiry A, Das D, et al. BootStomp: On the security of bootloaders in mobile devices. Paper presented at: 26th USENIX Security Symposium; 2017; Vancouver, Canada.
13. Bossi S, Visconti A. What users should know about full disk encryption based on LUKS. 2015. https://eprint.iacr.org/2016/274.pdf
14. Khati L, Mouha N, Vergnaud D. Full disk encryption: Bridging theory and practice. 2017. https://eprint.iacr.org/2016/1114.pdf
15. Müller T, Freiling FC. A systematic assessment of the security of full disk encryption. *IEEE Trans Dependable Secure Comput*. 2015;12(5):491-503.
16. Johnston SJ, Scott M, Cox SJ. Recommendations for securing Internet of Things devices using commodity hardware. Paper presented at: 2016 IEEE 3rd World Forum on Internet of Things; 2016; Reston, VA.
17. Hex-Rays. 2017; https://www.hex-rays.com.
18. Boursalian A. *BootBandit: A macOS Bootloader Attack* [master's project]. San Jose, CA: San Jose State University; 2017. http://scholarworks.sjsu.edu/etd_projects/559/
19. ASN.1 key structures in DER and PEM. 2017. https://tls.mbed.org/kb/cryptography/asn1-key-structures-in-der-and-pem
20. Google Cloud Platform. The Go development server. 2017. https://cloud.google.com/appengine/docs/standard/go/tools/using-local-server
21. Apple Inc. About system integrity protection on your Mac. 2017. https://support.apple.com/en-us/HT204899
22. Android: Verifying boot. 2017. https://source.android.com/security/verifiedboot/verified-boot
23. Apple Inc. iOS security. 2017. https://www.apple.com/business/docs/iOS_Security_Guide.pdf
24. Microsoft Inc. Secure the Windows 8.1 boot process. 2017. https://technet.microsoft.com/en-us/windows/dn168167.aspx
25. Aycock J. *Computer Viruses and Malware*. New York, NY: Springer US; 2006. Advances in Information Security.
26. Apple Inc. Startup security utility. 2018. https://support.apple.com/en-us/HT208198
27. Apple Inc. Secureboot. 2018. https://support.apple.com/en-us/HT208330
28. Apple Inc. Apple T2 security chip. 2018. https://support.apple.com/en-us/HT208862

**How to cite this article:** Boursalian A, Stamp M. BootBandit: A macOS bootloader attack. *Engineering Reports*. 2019;1:e12032. https://doi.org/10.1002/eng2.12032