EE8084 Cyber Security

Return-Oriented Programming to Compromise iPhone

Hailin Chen, Shengping Cui, Shuangchen Huang , Ziji Shi, Luoyuan Xiong and Shen Yuan

Nanyang Technological University

EE8084 Cyber Security

Content

Abstract

Return-Oriented Programming (ROP) is an advanced hacking technique with which hackers can break the normal sequence of control flow to take advantages of it without injection of any external code. It uses “buffer overflow” , a flaw in some programming languages including C, where a function did not check bounds before accepting data provided by user. Therefore, excessive data can overwrite the original return address of a function call in the stack, leading to a redirection to a new address specified by hacker other than the calling function. This technique is particularly effective for compromising iPhone [1]. We will begin by introducing principles of return-oriented programming, followed by a simple return-oriented programming program written in C for illustration. Then we focus on structure and characteristics of iOS system and a case study of compromising the iPhone with return-oriented programming. We show it possible to defense against return-oriented programming with proper settings [2]. Our conclusion calls for more awareness regarding the rise of return-oriented programming attacks.

Keywords: Return-Oriented Programming, iPhone, Information Security

Return-Oriented Programming to Compromise iPhone

1. Introduction

Return-Oriented Programming (ROP)was first proposed by *H. Shacham* in 2007 [3]. Making use of the C library, libc, which was pre-installed in almost every UNIX system, ROP combines small sequences of instructions into gadgets, which allows an attacker to perform arbitrary computation. ROP can easily bypass popular defense methods such as W⊕X by changing the return address to hijack the control flow. This ROP technique is particularly effective on x86 architecture since it is equipped with an abundant library where an attacker can easily find instructions to initiate an attack. Modular programming style is a commonly adopted programming style adopted by programmers who write large scale programs. However, modular programming also contributes to vulnerability of the program. After principles of ROP, a simple C program can illustrate the above basic ideas.

2. Literature Review

Modern computer systems are well-supplied with security features designed to protect them from being exploited by bugs. These safety methods may hide vital information or halt execution of a program when they detect suspicious behaviors.

Think about a case when an attacker who wants to hijack a computer system. To realize his goal, he must spot a vulnerability in some program and then make use of it. Exploitation means that he alters the program's control flow so that he could obtain the credentials of the program to do whatever he wants. The common vulnerability here could be the buffer overflow on the stack [7], though many other classes of vulnerability have been considered, such as buffer overflows on the heap, integer overflows, and format string vulnerabilities. In each case, two critical conditions must be satisfied [5]:

1. A technique is developed to change the program's control flow from its normal course.
2. The program is compromised to act in the manner the attacker chooses.

In traditional stack-smashing attacks [4], an attacker fulfills the first condition by overwriting a return address on the stack, so that it points to code of his choosing instead of to the calling function. He satisfies the second condition by inserting code into the process image; the modified return address on the stack points to this code.

Now we focus on how to fulfill the second condition. Defenders usually protect the vulnerable programs by preventing them from execution of the inserted code. In the earlier approach, the designers modify the memory layout so that it is impossible to execute the stack [6]. Since in stack-smashing attacks the shell code was typically inserted onto the stack, this was proven useful.

For instance, W⊕X, or called "Write XOR Executable", is a security feature firstly introduced by OpenBSD 3.3, released May, 2003. "W⊕X" means "write" and "execute" instruction can never be done at the same time on a given page, because many bugs are exploitable because the address space contains memory that is both writable and executable (permissions = W | X). Though it is sometimes a drag on performance, "W⊕X" were commonly deployed by Microsoft, Intel, and AMD.

Since the attackers cannot insert code, their response was to use code that already exists in the library. The standard C library, libc, becomes a target. This is because libc is preloaded in almost every Unix program and it contains routines of the sort that are useful for an attacker (e.g., wrappers for system calls) [5]. Such attacks are therefore known as return-into-libc attacks. However, theoretically any available code, either from the program's text segment or from a library it links to, could be used.

In the paper by H. Shacham [5], he proposed a thesis:

> *"In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation."*

In order to understand why ARM architecture based CPU is more vulnerable to attacks that are launched from memory than x86 based CPU, we need to learn more about their difference. ARM is a Reduced Instruction Set Computing (RISC) architecture while x86 being a Complex Instruction Set Computing (CISC) one. A reduced instruction set means it contains less number of instructions in its instruction set compared to x86. Therefore, ARM saves more power and silicon area on chip. However, the other side of the coin is that it takes more instructions to achieve the similar function as a complex instruction set will do. These features account for the reasons that ARM processor is most commonly found on mobile devices like iPhone, while x86 processor is more frequently employed on larger scale computers including desktops, servers and mainstreams. As mobile phones are becoming more frequently used in daily life than ever, it becomes a popular target chased by ROP attackers.

Another important difference between ARM and x86 architecture is that ARM is almost pure "Harvard Architecture" while x86 being nearly "Von Neumann Architecture". The former stores code and data on same place, while the latter physically store them on distinct areas. Therefore, ARM architecture can be more dangerous when exposed to ROP attack since the excessive data can easily overwrite the nearby codes.

Before we go into compromising iPhone, we need to learn about ROP in details. The following session will begin by introducing a commonly used top-down approach to write large programs—modular programming.

3. An Illustration of Return-Oriented Programming

3.1. Modular Programming and Stack

As size of computer programs is rapidly increasing, modular programming, which aims to separate a big program into a series of independent and interchangeable modules, is introduced and used widely. To achieve function calling and returning program flow, certain mechanism and method is applied. In this section, we introduce how is stack used in modular programming.

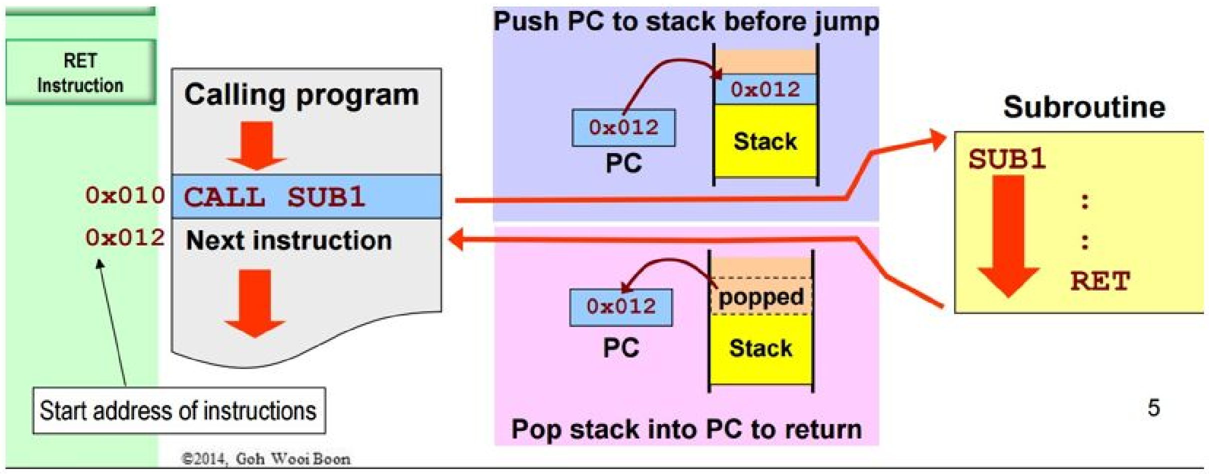


Figure 1. Module Programming on Stack

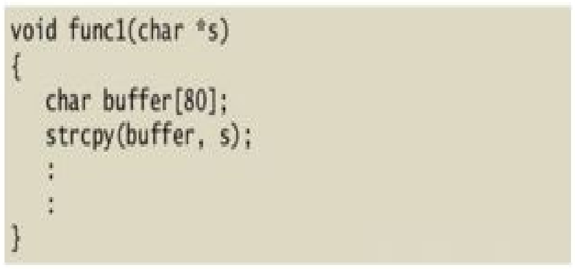
As figure 1.1 shows, when a module calls another module, the system will push the address of the next instruction (EIP) to be executed (0x012 in this case) to the memory stack. This is example shown by assembly code, but it is also applicable to C/C++ or other high level language's function calling as the compiler will either eventually translate these code into assembly code, or use this concept to produce binary code directly.

When the subroutine, or the function ends, the return instruction will direct the system to pop out the element on the top of memory stack (ESP) and deem it as the address of the next instruction to be executed (0x012 in this case). In this way, the program flow will go back to the main routine and execute the following code sequentially.

3.2. Basic Principle of Return Oriented Programming

Return Oriented Programming utilize the fact that EIP in the main routine will be stored in the memory stack and when function returns, the ESP will be deemed as the EIP address in the main routine. Using return oriented programming, hackers are able to change the content of the stack element which by right should represent the EIP in the main routine. In this way, the program will not go back to the main routine when a subroutine ends. Instead, it will wrongly think it is going back to the calling routine but end up directing to wherever the hackers want to the program to enter.

To realize this return oriented programming, we need to modify the content in the memory stack, which is fairly easy and possible in C language:



```
void func1(char *s)
{
   char buffer[80];
   strcpy(buffer, s);
   :
   :
}
```

Figure 2: *a C example*

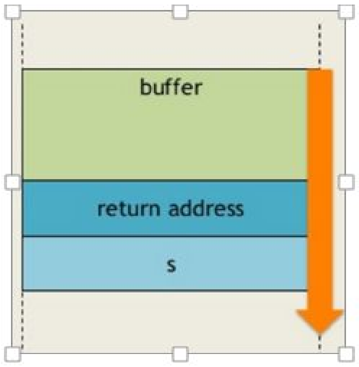


Figure 3: *Stack*

As shown in figure 1.2 and figure 1.3, the simple c function func1 receives one parameter called s with the type of pointer of character, and it create one local variable — as a string with the size of 80 character. when the other module calls this function, as shown in the stack graph, the system will firstly push the parameter s onto the stack, followed by the return address(the EIP of the calling routine), and create the local variable on the stack.

The next line of the function strcpy(buffer, s) simply tells the computer to fill in the buffer with the string that s points to (target). However, problems will occur when the size of the target is bigger than the size of the buffer. In this case, after the buffer is fully filled up, the strcpy function will go ahead to overwrite on the following stack until target has been fully copied, in which case the content of return address and even parameter s on the stack will be changed to part of the target. In addition, if we carefully design the content of the target, we can modify the return address and parameter with some meaningful data.

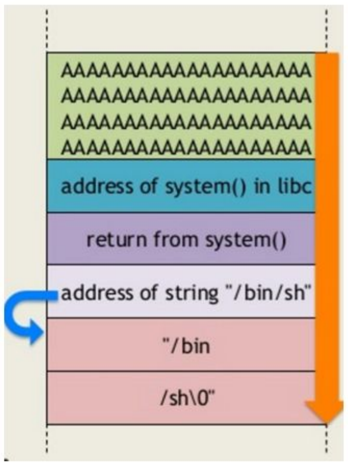


Figure 4: *Modified stack*

As shown in figure 1.4, the content of occupied stack area is modified in a smart way: the return address is overwritten by another function address. Therefore, the program will be directed to the function and the stack pointer (ESP) will then point to return from system(). When function system() executes, the program can even find the a parameter under ESP, which is address of string/bin/sh in this case.

3.3. ROP's limitations and solutions

With the ability to modify the content of the stack and thus direct the program flow, the most direct way of hacking a program is to write the hacker designed function in the stack and direct the program flow to that function. However, this is not feasible in practice as the stack area is marked as non-executable so even if hacker write attacking code into the stack, EIP will refuse to point to stack area.

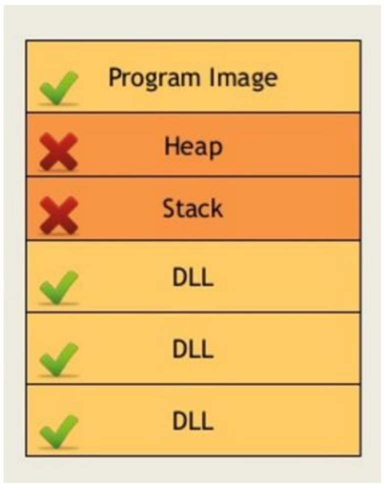


Figure 5: non-executable area

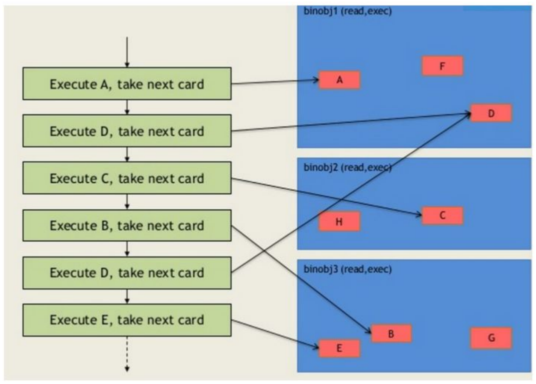

binobj1 (read,exec)
binobj2 (read,exec)
binobj3 (read,exec)
Execute A, take next card
Execute D, take next card
Execute C, take next card
Execute B, take next card
Execute D, take next card
Execute E, take next card
A
B
C
D
E
F
G
H

Figure 6: Orchestrating Code Execution

Therefore, return oriented programming uses another way—Orchestrating Code Execution. As there are a lot existing codes in the executable memory—usually code pieces in the binary or shared library. The program can perform complex operations by calling a series of code pieces.

4. Introduction to iOS Security Architecture

iOS is currently the second most popular mobile operating system, and it is the sole OS that Apple Inc.'s iPhones are shipped with. Long known for its simplicity and outstanding graphic user interface, the recent San Bernardino case, in which Apple opposes the order of the FBI to hack into a suspect's iPhone, brought forward the security mechanisms applied by iOS. Nowadays the general public becomes increasingly aware of the importance of data security with regard to mobile devices, and this part of the report will give a brief introduction to iOS and its security features.

4.1. iOS

iOS, originally named iPhone OS, is the mobile operating system developed by Apple Inc., and was unveiled in 2007 along with the original iPhone. The development of iOS, led by Scott Forstall, is based on the Mac OS desktop operating system. Though like Mac OS, iOS runs on Unix-like XNU hybrid kernel and Darwin foundation, there are many differences between them, including the specifically defined multi-touch gesture interface and restricted Unix-like shell access on iOS. In October, 2008, Apple released the first Software Development Kit (SDK) for iPhone OS, which allows developers to build third-party applications that behave like native apps in the Xcode development environment. Users can only download third-party apps from App Store, Apple's mobile app digital distribution platform, unless the device is configured to accept unauthorized apps.

4.2. Understanding the Threats

iOS devices, like desktop computers, face a various of threats. In general, the threats can be divided into two categories: malware and exploits. And by elaborating the threats, we can better see the intention behind the design of iOS security architecture.

Malware is a piece of software that intent to steal and damage information, or spy on users without their knowledge. Malware can hide itself in other software or disguise itself as benign software, and when executed, it would perform malicious operations, such as planting advertising software and authorizing remote access to the attacker. It has been a traditional threat for personal computers and now it endangers mobile devices as well. Once executed, the computer is relatively vulnerable because they are designed to run software and machines rely on anti-virus software's objective to identify the malware.

Exploits, different from malware, take advantage of the bugs and vulnerabilities of existing software to cause unanticipated behavior or gain control of the computer system. It is generally considered a bigger threat than malware as it does not require the user to install any software: with one rigged pdf document, web page or email, attacker can gain control of the computer system without any knowledge of the user. To protect the computer from exploits, two approach can be exercised. The first way is to eliminate the flaws of the code as much as possible, and hide the source code which makes it difficult for attackers to find a flaw they can make use of. The second way is to cut the connection between flaws and malicious code. The corresponding method, including memory randomization and data execution prevention will be explained in the next section.

With the knowledge of iOS and its major threats, we can look deeper into its security architecture.

4.3. Security Architecture

4.3.1. A “Crippled” System

An attacker surface is the software environment where attackers can try to input data or draw data from the environment. As mentioned above, one practice to protect the system form exploits is to minimize the amount of code accessible to unauthorized user, and Apple’s iOS has a smaller attack surface compared to other mobile platforms, or Mac OS. One common complaint about iOS is that it does not support Java and Adobe Flash. Apple has a good reason to do this because Java and Flash are well-known for their security vulnerabilities, and since the new HTML standard support most of their features, ditching these two application appears to be a wise move. Moreover, many file formats, common or not, are not recognized by iOS or its mobile web browser Safari. Users can handle these files using third-party applications running in a separate system layer which substantially avoids the risk. For instance, computer security researcher Charlie Miller, in his exploitation on Preview, discovered over a hundred crashes. And when he tried the same technique on iOS, only 7 percent of them really caused problems. Other than reducing attacker surface, Apple removed shell form iOS. Shell is an essential component of Mac OS, as it enable Macintosh machine to possess both an exquisite easy-to-use GUI and UNIX environment. Shell is also the ultimate goal of exploits: to execute a shell and command utilities from it. With the absence of shell in iOS, the attackers lost access to those tools and they may be forced to incorporate all their actions inside the exploit code, which is not easy to tackle. Albeit many are unsatisfied with the limitations of iOS's functionality, what most people want from a mobile OS is simplicity and reliability, and Apple did weigh the gain and the loss.

4.3.2. Privilege Separation

Privilege Separation is a basically to separate a program into parts with various levels of privilege, which are required to perform certain task. This technique is generally used for damage control when an attack is successful. In iOS, processes are divided by UNIX permission mechanisms: crucial system processes run as privileged root, while all third-party applications, along with some native app like the Safari browser, runs as the user mobile. With the help of this structure, even if an attacker get his code run through flaws in some third-party app, the attacker's code could only run as user mobile, and such successful exploit cannot change important system configuration.

4.3.3. App Store: the Anti-virus Software of iOS

As mentioned before, to protect the device from malware, anti-virus software is needed to identify malware and delete it before executing it. However, instead developing a dedicated anti-virus software, Apple uses Mandatory Code-Signing, which is one of the most important parts of iOS security architecture. Code signing means all source code files must be signed by authority, otherwise the system will not execute them. In iOS, the authority is Apple, and the signing process is implemented by App Store, the only platform where regular users can download third-party applications.

After an app is built by developers, it is sent to Apple for review and if approved, it will be signed by Apple's private key and then make its way to App Store for download. In this review process, Apple plays the role of anti-virus software. It examine each app and determine if they are safe to run and if the software can update and rewrite itself. There are currently more than 1.5 million apps available on App Store, and there are few cases where malware managed to get through the review process.

4.3.4. Data Execution Prevention on iOS

Data execution prevention is a mechanism which mark portions of memory as non-executable, so the attempt to execute code hiding in data region will cause an exception. Attackers bypass DEP by Return Oriented Programming, in which attackers use pieces of legitimate code to create a piece of memory in stack and heap area where ROP is not enforced, thereby execute their code written elsewhere without being noticed. However, in iOS, code signing prevent the attackers from using ROP to run their unsigned code in storage. In other word, it is almost impossible to find a section of memory where DEP is off. As a result, exploits need to be carried out completely by ROP, which is simply difficult.

4.3.5. Address Space Layout Randomization

Another component of iOS defense is Address Space Layout Randomization (ASLR). Randomization of memory address further reduce attackers' chance to perform a successful ROP because they need to know the address of the legitimate code they are taking advantage of. In iOS, stack and heap memory addresses, along with the location of libraries and linkers are all randomized. With the help of DEP and ASLR, attackers must found both a vulnerability to control code execution and a leaked memory address to implement ROP. Consequently, there is a slimmer chance of critical flaws detected by attackers.

4.3.6. Sandboxing

While ASLR and DEP pose a seemingly insurmountable obstacle on the attacker's path, damage control is still needed in case the attacker did managed to get code execution. To do this, iOS introduces sandbox, which is similar to privilege separation, but is specially tailored for each applications. Sandbox pose a more restricted environment, insulating an app from the resources it won't use in nature, or other apps. For example, a mobile game software would not need access to user's contacts, so the system create a unique interface for the app by which it cannot get information the contacts. By doing so, the apps' normal functionality will not be affected, and the potential danger is avoided. In addition, if a piece of malware is executed, or the execution control the attacker can only access what the interface offered, and the damage is well-bounded inside the sandbox.

From all the information above, we can conclude the design principles of iOS security architecture: reducing attacker surface, process isolation and damage mitigation. Basing on which, the various techniques and security policies Apple adapted makes intrusion of malware and exploits extremely difficult, though not impossible.

5. Case Study

5.1. Jail Breaking

One of the implements of Return-Oriented Programming is to jailbreak iOS. The term jailbreaking refers to the process of removing operation system safety limitations imposed by the manufacturer. Jailbreaking does not refer to iOS jailbreaking explicitly, but many other devices can also be jailbroken. The most practical reason for this tough process is to expand and unlock the otherwise limited features set imposed by Apple's code signing security for all the applications present on the device. In other words, jailbreaking is basically trying to make adjustments to the operating system on the hardware resulting in semi-permanently disabilities of Apple's certificate security mechanisms, allowing any third-party software (code) to run on the device. End user jailbreaking are the most common form of jailbreaking, and are designed to benefit the end user by enabling him to access the device and install third-party software. End user jailbreaks are written for general purpose, often performed by one of the popular jailbreaking tools available. They often also install new applications, which appear on the device's home screen, making it apparent to anyone looking at the device that it has been jailbroken. Many publicly available jailbreaking tools additionally integrate at least one software installer application, in this case of JailbreakMe, the one we are going to talk about right after, is

Cydia, which allows end users to run tools and software from a third-party online file expository. There are other more customized forms of jailbreaking using less detectable jailbreaks, which may be performed by spyware, malware, or through intentional covert hacking. In these cases, it may not be so apparent that the device has been jailbroken because application icons may not have been added to the screen. Over the history of Apple's iPhone devices and iOS platform, many jailbreaking tools have found their way into the mainstream, and also the open source community has built a large sum of third-party software, available both freely and commercially through these installers. Much of these software has not or would not pass the permission with Apple's strict App Store policies.

Upon the iPhone's initial release, hackers began to notice that the iPhone operating system ran in a jailed environment. The term jailbreaking originated from the very first successful iPhone hacking to break out of this jailed environment, allowing files to be read and written anywhere on the devices. A jailed environment is an untrusted environment subordinate to the standard user environment, imposing additional limitations on what an end user can access. That means end user's accessibility is limited by application sandboxing and developer code-signing, as well as restrictions placed on iTunes. The iTunes only permitted applications to access only certain files on the devices—namely those within a jail rooted in the Media folder on the devices. Developers started to realize that it would be critical to have the abilities to read and write anywhere onto the hardware for attaining further access and more control over the device. This led to the world's first third-party iPhone applications, long before the SDK (software develop kit) or App Store ever existed.

JailbreakMe is a series of jailbreak tools for Apple's iOS system, which take advantage of vulnerabilities in the Safari browser, featuring an immediate one-step jailbreak while most other jailbreaks require plugging the device into a computer and running the jailbreaking software from the computer. Jailbreaking allows end users to install software that is not permitted by Apple on their App Store. JailbreakMe automatically installs Cydia, a package management software that serves as a replacement to the App Store. JailbreakMe's first public version in 2007 worked on iOS firmware 1.1.1, the second version came out in August 2010 for firmware 4.0.1 and earlier, and the third version was released in July 2011 for iOS versions 4.3 to 4.3.3, and was the first jailbreak for the iPad 2. At least two million devices have been jailbroken by JailbreakMe 3.0.

First in 2007, the hackers at the Jailbreakingme.com found that the iOS MobileSafari installation had vulnerabilities within their implementation of a C library, called FreeType, which only functions as a font rasterization engine for rendering text to bitmaps, contained one critical flaw. When it rendered a specially crafted PDF it would end up with a memory stack overflow condition which then resulted in a privilege override. This allowed the rest of the malicious code attached to the PDF to call some of its own otherwise benign looking libraries, resulting in the final exploit code being loaded and executed. In the case of Jailbreakme.com the code loads Cydia, as the illustration down below. This jailbreaking process can be done remotely, without any USB connection or anything, but only an Internet connection, which absolutely grabbed Apple's attention as soon as possible.

***Figure** 7:Jailbreaking method 1*

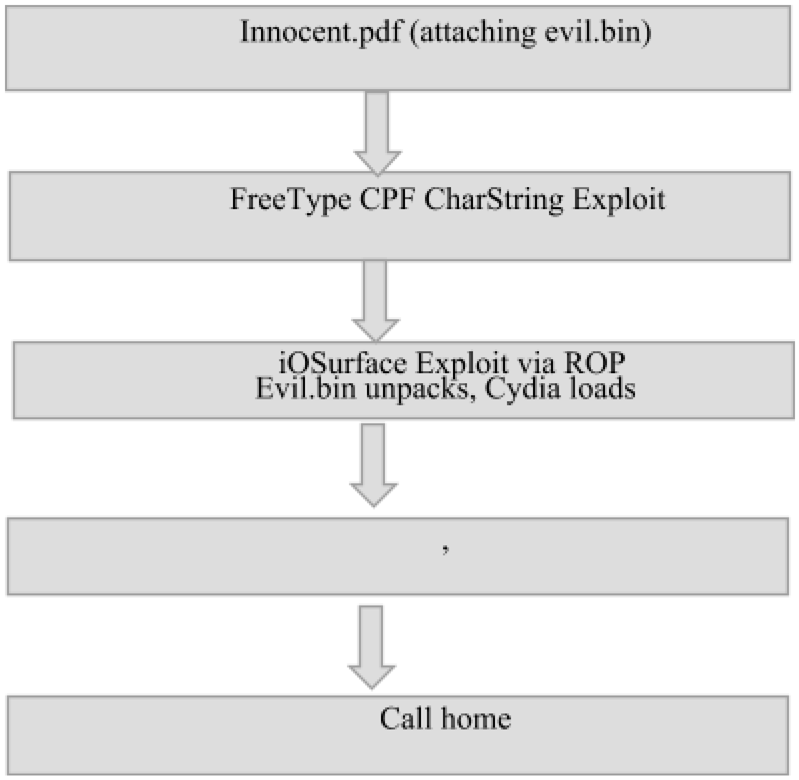

Innocent.pdf (attaching evil.bin)
FreeType CPF CharString Exploit
iOSurface Exploit via ROP
Evil.bin unpacks, Cydia loads
,
Call home

Figure 8: *Interface of Cydia*

The resulting Return-Oriented Programming allows the code to attack a kernel memory vulnerability inside of iOSurface resulting in the setuid being set to 0(root). With the new root privilege, the evil.bin file embedded within the innocent PDF can be unpacked, and executed by the .lib files to install whatever packages the attackers want behind the scenes.

In iOS 2.0, Apple decided to further secure their system by implementing data execution prevention (DEP), much like the versions in the newer OS X and windows. DEP is required for all applications running within iOS, which makes it extremely difficult to achieve arbitrary code execution. But a ROP payload can get around this restriction by connecting the already present code in the system to attempt to try gain base access.

In most cases on the workstation operating systems it's possible to disable DEP using a ROP code, which then gives you the ability to write normal shellcodes to the stack. Unfortunately for attackers, there was no known way to disable the code signing in the iOS.

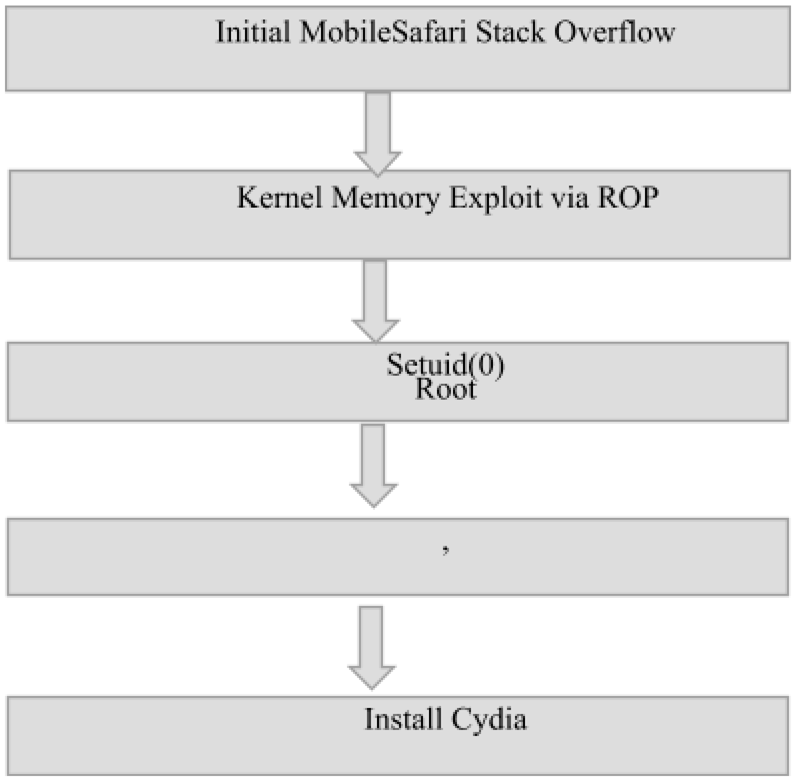

Initial MobileSafari Stack Overflow
Kernel Memory Exploit via ROP
Setuid(0)
Root
,
Install Cydia

Figure 9:*Jailbreaking method 2*

This resulted in attackers having to write the entire payload in ROP, which is particularly time-consuming, or use a two-stage attack for userland and kernel space, like jailbreakme.com did.

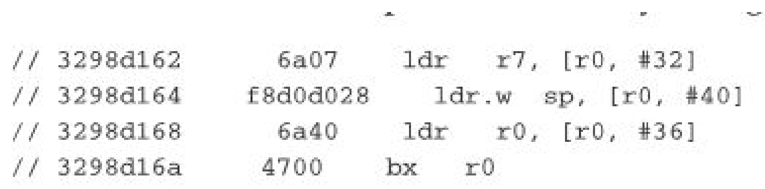
In iOS 4.3 and higher versions, it turns out that Apple wants to patch this vulnerability, as they introduced the Address Space Layout Randomization mechanism. ASLR makes it further more difficult to interact with the memory for exploit since we cannot possibly predict the address of the memory stack and heap any more. As a result, any exploit attempting to use ROP technics needs to discover the base address of a memory module. This further challenged the methods used to get arbitrary code execution on the iOS devices. The hackers at jailbreakme.com had to use another vulnerability, found in the iOSurface, which allowed them to manipulate kernel memory and setuid (0) via ROP, as the illustration down below. What the attacker is trying to do is to use ROP to chain two different exploits together, a remote one and a local one. Two exploits open a PDF file in the MobileSafari browser: initial code execution is obtained through a vulnerability in the Freetype Type 1 font parser, allowing subsequent exploit of a kernel vulnerability to disable code signing enforcement, get root access and install the jailbreak. The same kernel vulnerability is also exploited at each reboot to provide an untethered jailbreak, using the incomplete code-signing technique to bootstrap the kernel exploit. By doing this the attackers can get around the userland code signing and execute a normal payload in either kernel space or userland.

5.2. SMS Database Exfiltration

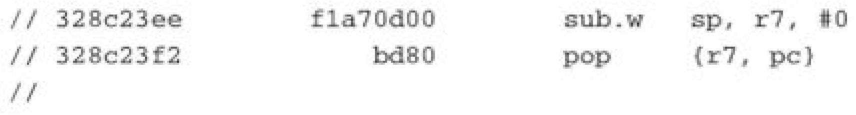
Apart from the famous jailbreaking application introduced above, another good real-life example iOS exploitation using ROP-only shellcode is Exfiltration of content on SMS database. This ingenious method was first demonstrated publicly on PWN2OWN competition by Ralf-Philipp Weinmann and Vincenzo Iozzo in 2010. For the record, PWN2OWN is an annual computer hacking competitions held for the purpose of showcasing newly discovered or unknown vulnerabilities.

The so-called “Exfiltrate File Content Payload” aligns inherently with binaries from iOS 3.1.3 for iPhone 3GS. The underlying principle is of no difference compared to a normal ROP Shellcode, and a brief illustration of this method is shown below.

Taking control of the stack pointers and multiple registers is the first action on the list. Recognizing that the only controllable register, at the beginning of the shellcode execution, is R0, pointing to a buffer with 40 bytes in length.

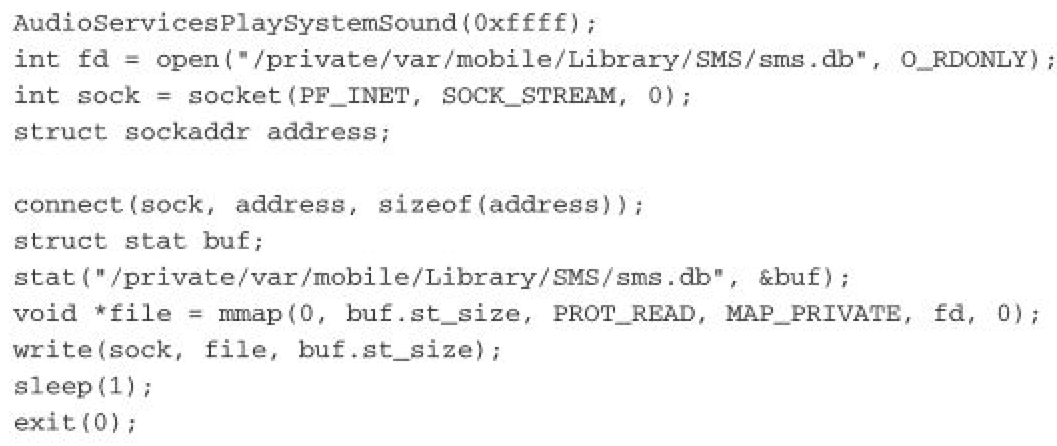

```
// 3298d162      6a07     ldr   r7, [r0, #32]
// 3298d164    f8d0d028    ldr.w  sp, [r0, #40]
// 3298d168      6a40     ldr   r0, [r0, #36]
// 3298d16a     4700    bx   r0
```

Having R0 under control, the payload then points R7 to another controlled location. Another gadget is needed as stack pointer, pointing to random memory, is past the 40 bytes under attacker control. Here follows the second gadgets that serves such purpose.


```
// 328c23ee        f1a70d00        sub.w   sp, r7, #0
// 328c23f2            bd80        pop     {r7, pc}
//
```

By redirecting what's inside of R7 into the stack pointer and setting the stack to an attacker-controlled location, the attack could essentially operates ROP payloads from which the instruction pointer is extracted.

Then the payload tends to open up the SMS database and a socket by calling a function that vibrates the device for debugging. Consequently, function stat() and mmap() are invoked to get the size of the file and enable file sending to remote server respectively. Interestingly, the original payload utilize sleep() to avoid untimely disconnection to the serve before completion of file transportation. To better elaborate, the following operations, written in pseudo-C, will be executed to achieve the above steps.



```
AudioServicesPlaySystemSound(0xffff);
int fd = open("/private/var/mobile/Library/SMS/sms.db", O_RDONLY);
int sock = socket(PF_INET, SOCK_STREAM, 0);
struct sockaddr address;

connect(sock, address, sizeof(address));
struct stat buf;
stat("/private/var/mobile/Library/SMS/sms.db", &buf);
void *file = mmap(0, buf.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
write(sock, file, buf.st_size);
sleep(1);
exit(0);
```

With all that, the elegantly yet simply designed payload realizes the aim of stealing SMS information from its database, using basic ROP idea.

6. Conclusion

We have presented a way named Return-Oriented Programming (ROP) to change control sequence so that an attacker can initiate arbitrary attack. The main stage is to construct short sequences of instructions using a library that exists in the memory. The trick is that normal security method can hardly detect such attack since it purely uses legitimate instructions. Then the return address can be modified so that the program control falls into the attacker.

iOS is one of the most popular and secure mobile systems. It applies various security method to increase its reliability, including reducing attacker surfaces by privilege separation and sandboxing. Besides, its memory is protected by address space layout randomization.

However, ROP helps bypass iOS' security methods. Attackers can use a crafted PDF to cause stack overflow when the iOS' preinstalled browser, Safari, previews the PDF file. By doing so, their iOS devices can be "jailbroken". This enables to user to change arbitrary settings, which is not allowed by Apple. However, jailbreaking exposes users to the danger of leak of personal information.

To protect from ROP attacks, frequency of instruction usage is expected. By analyzing suspicious app instruction sequence and comparing it with normal sequence, we can detect the unusual gadget that is used to alter control flow. Automation of this detection process is promising in future research area of mobile security.

References

[1] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A. R., Shacham, H., & Winandy, M. (2010, October). Return-oriented programming without returns. In Proceedings of the 17th ACM conference on Computer and communications security (pp. 559-572). ACM.

[2] Davi, L., Sadeghi, A. R., & Winandy, M. (2009, November). Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In Proceedings of the 2009 ACM workshop on Scalable trusted computing (pp. 49-54). ACM.

[3] Miller, C. (2012). *iOS Hacker's Handbook.*

[4] Pincus, J., & Baker, B. (2004). Beyond stack smashing: Recent advances in exploiting buffer overruns. Security & Privacy, IEEE, 2(4), 20-27.

[5] Shacham, H. (2007, October). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and communications security (pp. 552-561). ACM.

[6] Wojtczuk, R. (1998). Defeating solar designer nonexecutable stack patch.

[7] Zhang, C., Wang, T., Wei, T., Chen, Y., & Zou, W. (2010). IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In Computer Security–ESORICS 2010 (pp. 71-86). Springer Berlin Heidelberg.

[8] *Analysis of the jailbreakme v3 font exploit*. (2011). Retrieved from http://esec-lab.sogeti.com/posts/2011/07/16/analysis-of-the-jailbreakme-v3-font-exploit.html

JailbreakMe Wikipeia. (n.d.). Retrieved from https://en.wikipedia.org/wiki/JailbreakMe

[9] *Technical Analysis on iPhone Jailbreaking*. (2010). Retrieved from http://community.websense.com/blogs/securitylabs/archive/2010/08/06/technical-analysis-on-iphone-jailbreaking.aspx