```
.17728 FB size=3145728, FIFO size=2097152
3200 : host_bpp=32, bpp=32, num_displays=2
58 bpp=32
J_CSTData - _CST evaluation failed

ca63: Info 0xffffff7f81490060 Name com.vmware.kex
built Oct 31 2012 17:00:26

04...

e display is awak
e display is awak
rsorImage() failed
920x1200, 32 7680
```

```
free_to_zone(zone_t zone, void *elem) {
        unsigned int i = sizeof(vm_offset_t) /
        zone->free_check_count++ % free_check
        zone->elem_size >= (2 * sizeof(vm_off
                zone->free_check_count = 1;
                ((uint32_t *) elem)[i] = ZP_POISO
                for (i++; i < zone->elem_size /
```

You need to restart your computer. Hold down the Power button until it turns off, then press the Power button again.

Redémarrez l'ordinateur. Enfoncez le bouton de démarrage jusqu'à l'extinction, puis appuyez dessus une nouvelle fois.

Debe reiniciar el ordenador. Mantenga pulsado el botón de arranque hasta que se apague y luego vuelva a pulsarlo.

Sie müssen den Computer neu starten. Halten Sie den Ein-/Ausschalter gedrückt bis das Gerät ausgeschaltet ist und drücken Sie ihn dann erneut.

コンピュータの再起動が必要です。電源が切れるまでパワーボタンを押し続けてから、もう一度パワーボタンを押します。

XNU: a security evaluation

Certified Secure

UNIVERSITY OF TWENTE.

XNU: a security evaluation

D Keuper (s1019775)
University of Twente and Certified Secure

December 13, 2012

Abstract

The XNU kernel is the kernel that powers Apple's OS X and iOS operating system. Originally developed by Next, but later acquired by Apple it has been around for more then twenty years. It is only since the introduction of code signing on iOS that hackers have developed a real interest in XNU. Prior to iOS 2.0 only a handful of articles were published on XNU in the context of security. In the past few years the interest of hackers increased by the popularity of jailbreaking (the process of removing the security restrictions of iOS) and iOS security is often discussed at the various security conferences. The interest of hackers in the XNU kernel has not gone unnoticed by Apple, which started to harden the security of their XNU kernel. In the latest releases of their operating systems they added different mitigation techniques such as kernel ASLR.

The latest release of OS X (Mountain Lion) and iOS (6) are hardened with new protection mechanisms, which should reduce the risk of successful exploitation. In this thesis we describe all techniques that are meant to protect the kernel from attackers and their implementation. We argue the effectiveness of those techniques and how an attacker might try to circumvent them. We conclude this thesis with a comparison with other kernels, our opinion on the current security state of the XNU kernel and some recommendations on how to further improve the security.

We state that the XNU kernel has many protection mechanisms in place, but that they could be improved to better protect against exploitation. Furthermore we state that the current design of XNU, which consists of two separate components that heavily interact, is error prone and would benefit from refactoring.

Contents

Part I

Introduction

1 Introduction

1.1 Introduction

XNU, which is an acronym for X is Not Unix, is the kernel used by Apple in its OS X and iOS operating system. It has been the default kernel for Apple's operating systems since the first release of Mac OS X in 2001. Originally developed by NeXT for their NeXTSTEP operating system (which was later acquired by Apple), the XNU kernel has a long history with its first public release in 1989.

For this thesis we investigated the current state of the XNU kernel in the context of security. We try to answer the question 'how secure is the XNU kernel?'.

1.2 The history of XNU

The Mach kernel project was started in 1985 at the Carnegie Mellon University in Pittsburgh. Its goal was to build a true micro kernel which could act as a replacement for the BSD kernel. At the time the generally accepted theory was that micro kernels would eventually replace the monolithic kernels used at the time. In a micro kernel design as much code as possible is moved from kernel space to user space in so called services, making the kernel a message parsing service between the different services and between the hardware and its drivers in user space. This design should improve the overall stability since crashed user space services could be restarted without panicking the kernel, as well as offer a better security model by running services, such as drivers, under a less privileged account.

While the design of a micro kernel is favorable, there were considerable performance issues. The first versions of the Mach kernel still contained the entire BSD kernel and was thus far from being micro. The idea was to slowly phase out the BSD kernel components and move them to user space, in separated services. Since the different services should be able to communicate with each other, Mach implemented a Inter Process Communication (or IPC) system, which allowed user space applications to send messages to each other. However, because for each message multiple context switches were needed (from user space to kernel space and back to user space) and the access rights had to be checked, performance were poor. For instance, a dummy syscall on BSD would take 40 μs where its Mach counterpart would take nearly 500 μs[20]. Due to these performance problems the project was canceled in 1994.

Despite the performance problems the Open Software Foundation (OSF) announced they would use the Mach 2.5 kernel in their OSF/1 operating system and NeXT adopted the Mach 2.5 kernel for their NeXTSTEP operating system. However, the Mach kernel was not yet finished, it was lacking some important features like a virtual file system and a network stack. Therefore NeXT copied these features from the BSD kernel. This mixture of Mach and BSD was called XNU. Since the BSD components also run in kernel space, the XNU kernel is not a micro kernel but a monolithic kernel.

In the nineties Apple had to admit that their Mac OS operating system was missing some important features which were needed to be able to compete with their competitors at the time, most notably Microsoft who had just released their Windows 95 operating system. Apple's current operating system at the time, called System 7, had no support for protected memory and preemptive multitasking. For this reason Apple started a new research project called Copland. Copland was intended to become the successor of System 7, but the release date kept slipping into the future. It became clear to Apple that Copland would never see a final release, so Apple decided to buy an existing operating system instead, and cancel the development of Copland.

There were two candidates at the time for an acquisition by Apple. Both candidates were founded by ex employees of Apple. The first option was Be which developed BeOS and was founded by Jean-Luis Gassée. The other was NeXT, which was founded by Steve Jobs, with their NeXTSTEP operating system. Apple chose for NeXT and announced the acquisition of NeXT on December 20, 1996.

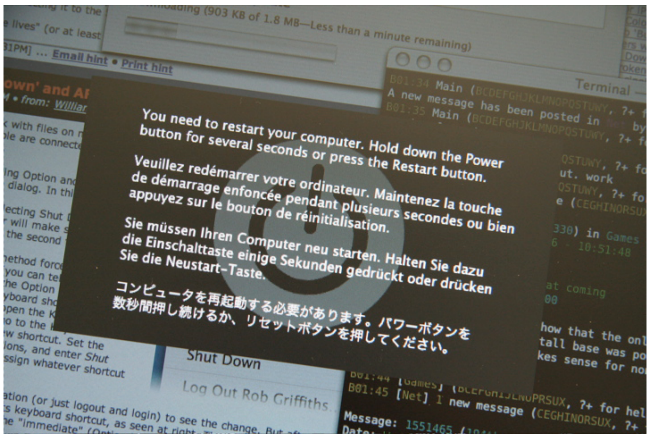
Ever since the acquisition of NeXT, Apple continued working on their new operating system under the code name Rhapsody, which would eventually be released as Mac OS X (which was renamed to just OS X with the release of Mountain Lion). Before the consumer version was released, Apple first released Mac OS X Server 1.0 on March 16, 1999. This server OS contained components from all of the operating systems Apple owned at the time (Mac OS, NeXTSTEP and Mac OS X). The consumer edition, called Mac OS X 10.0, was released on March 24, 2001, more then four years after the acquisition of NeXT. Mac OS X was the first operating system by Apple that featured the XNU kernel, and it has been the OS X kernel ever since. Apple made some changes to the original XNU kernel over time, most notably the additions of I/O Kit for driver development, the update of the Mach component from version 2.5 to version 3.0 and the porting of XNU to the Intel and ARM processors as well as the transition from 32-bits to 64-bits.

In 2007 Apple released the first iPhone, which ran on Apple's iOS operating system (previously named iPhone OS, but renamed to iOS since version 4.0). The iOS operating system is in its core equal to OS X with a different graphical interface to support the multi touch interface. The core of iOS is almost equal to that of OS X, meaning it also runs on the XNU kernel. Apple made some additions to the XNU kernel to support code signing, a feature that would later see the light in OS X as Gatekeeper.

1.3 Kernel security and exploitation

Kernel exploitation is the art of exploiting a vulnerability in kernel space. It is no new technique and kernel exploits have been seen for all major platforms. Nonetheless they are less common than exploits for user space applications, which can be devoted to the fact that kernel exploits often require a higher skilled attacker due to the added complexity in kernel space, by for example concurrency and a less sophisticated debug setup.

The performance of today's computers allows us to implement more advanced exploitation mitigation techniques. Modern operating systems are equipped with multiple techniques in order to prevent successful user space exploitation, such as NX protection, ASLR and stack canaries. While many of those techniques are provided by the kernel, it is interesting to note that they are often not used to protect the kernel itself. This can be because the impact on performance would be too large, or because of implementation difficulties in kernel space. This makes the kernel an interesting target for attackers and protecting the kernel against those attackers is critical. The kernel runs at the highest privilege level available and exploiting the kernel gives the attacker full system privileges. As 'a chain is only as strong as its weakest link, which will become the target for attackers'[36]. If user space applications are protected by advanced exploitation mitigation techniques, it is only natural that attackers will focus on the kernel instead.


You need to restart your computer. Hold down the Power button for several seconds or press the Restart button.

Veuillez redémarrer votre ordinateur. Maintenez la touche de démarrage enfoncée pendant plusieurs secondes ou bien appuyez sur le bouton de réinitialisation.

Sie müssen Ihren Computer neu starten. Halten Sie dazu die Einschalttaste einige Sekunden gedrückt oder drücken Sie die Neustart-Taste.

コンピュータを再起動する必要があります。パワーボタンを数秒間押し続けるか、リセットボタンを押してください。

Figure 1.1: a crashed XNU kernel

The downside of kernel exploitation is stability. Since the system only runs because of the kernel, crashing it is not an option. This is different from user space exploitation, where brute forcing is sometimes used to bypass one of the security mitigation techniques, since restarting the application is often not a problem. Brute forcing in kernel space often means leaving the kernel in an unstable state which could lead to a panic. All memory that an attacker overwrites during exploitation has to be restored afterwards, otherwise the kernel could crash at a later point in time. This requires a reliable exploit which, depending on the vulnerability, is not always possible.

Because the kernel should remain in a stable state, the attacker should have a deep understanding of some of the kernel internals. There are more processes that require kernel resources at the same time. Therefore a solid understanding of the kernel scheduler and virtual memory system is required for writing a reliable kernel exploit.

1.4 Contents

1.4.1 Purpose of our research

Millions of people use the XNU kernel everyday without knowing. The XNU kernel is used to power almost all hardware platforms from Apple. All OS X devices like the MacBook and iMac are powered by XNU, but also all iOS powered devices, such as the iPhone, iPad and the Apple TV, run on the XNU kernel. During the lifetime of XNU it has powered numerous operating systems and got ported to multiple architectures like the PowerPC, Intel and ARM.

While the XNU kernel is roughly twenty years old and is used on so many devices in today's world, it never received much attention from the security scene. It is only since the release of iOS 2.0, which featured code signing, that hackers (or for iOS often referred to as jailbreakers) started investigating and exploiting the XNU kernel. Because the jailbreaking scene is playing a cat and mouse game with Apple by breaking the security model of iOS, Apple is starting to invest into kernel security.

Since the XNU kernel was long time ignored by the security field a lot of information about XNU in the context of security is either missing, incomplete or outdated. With our research we hope to close this gap and provide other researchers with a current list of security features in the XNU kernel and new insights about exploiting it. Our research is threefold, first we will discuss some of the current features of the XNU kernel in the context of security. This covers both the implementation details of certain components as well as the currently active mitigation techniques. Secondly we will discuss what attackers can do to gain code execution in kernel space and thus circumvent the present mitigation techniques. Lastly we compare the XNU kernel with other kernels and their mitigation techniques.

1.4.2 Research questions

We try to answer the following research questions:

1. **What security mechanisms are currently implemented and how effective are they against exploitation?**
   Questions like how effective is the XNU kernel ASLR implementation, is the kernel also protected by NX etc. are currently left unanswered. Answers to those questions are of great influence of the level of protection a kernel has against attackers.

2. **How much protection does the XNU kernel offer when comparing it to other kernels?**
By comparing the answer of the first research question with that of other mainstream kernels, we can say something about the level of security the XNU kernel offers to its users, as well as pointing out some points of improvement.

3. **Which exploitation techniques can be used under XNU?**
When looking at other kernels there are a lot of public exploits available together with presentations that cover the topic of exploitation. This not only makes the life of exploit developers a little easier, but this information is also useful for further protecting a kernel against attackers. By investigating these techniques, developers can focus on disarming the commonly used tricks.

4. **How can we harden the security of the XNU kernel?**
By answering the three research questions above we should have a solid understanding of the current state of security in kernels and the XNU kernel in specific. Based on those findings we should be able to make some specific recommendations for further enhancing the security of the XNU kernel.

1.4.3 Versions and source code

This research is based on the latest version of OS X and iOS at the time of writing. For OS X this is version 10.8.2 (Mountain Lion) and for iOS this is version 6.0.1.

Throughout this thesis numerous source code examples are listed from the XNU source tree, these were taken from XNU version 2050.18.24. A local copy can be obtained from opensource.apple.com/tarballs/xnu/xnu-2050.18.24.tar.gz or be viewed on line at opensource.apple.com/source/xnu/xnu-2050.18.24/. The XNU source is released under the BSD[1] and APSL[2] license.

The core component of OS X and iOS is called the Darwin operating system and is maintained by Apple under an open source license. Darwin is POSIX compliant and contains the XNU kernel as well as many system libraries and system utilities. Darwin does not contain the graphic components that are used in OS X and iOS. More information about Darwin and its source code can be found at Apple's open source website at opensource.apple.com.

1.4.4 Outline

We start with describing the design an implementation of the XNU kernel in Chapter 2. This includes a description of the different components of XNU, the syscall calling mechanisms and how XNU manages memory allocations. This chapter concludes the introduction part of his thesis.

[1]opensource.apple.com/license/bsd/
[2]opensource.apple.com/license/apsl/

The next part focuses on the security mechanisms present in todays XNU release. In Chapter 3 we describe all techniques that should mitigate the risk of memory corruption in kernel space (e.g. kernel ASLR and heap protection). Chapter 4 describes the more advanced protection mechanisms, that not only protect the kernel but the whole system (i.e. code signing and sandboxing). This part will only describe the implemented techniques, but not how an attacker might bypass them. Note that we can only see the protection mechanisms that should prevent exploitation at runtime. It is unknown if Apple also uses some form of static or dynamic source code analyzer to detect vulnerabilities before compilation.

Part III focuses on security research and exploitation. We start this part with Chapter 5, which describes the techniques that can be used to find new vulnerabilities as well as possible attack vectors. Chapter 6 goes into greater detail on how to bypass the mitigation techniques and exploit certain vulnerabilities (such as exploiting the zone allocator). Knowing where and how an attacker might strike, helps in forming the recommendations to increase the level of protection the XNU kernel can offer.

In the final part we compare and conclude our research. Chapter 7 compares the security features of XNU with that of other kernels. In the final chapter (Chapter 8) we answer our research questions and give our opinion and recommendations about the security of XNU.

A toolchain for kernel research is described in Appendix A. This appendix will cover both the steps required for debugging a live kernel as well as the tools required for vulnerability research.

2

General design

2.1 Introduction

The XNU kernel is a hybrid of the Mach kernel and the BSD kernel. There are no user space services (for the file system or network stack for example) like in the design of the original Mach kernel. This has its implications for security, since the attack surface for the kernel is much larger in a monolithic design. Kernel services that would run in user space could run as a less privileged user, something which is not possible in a monolithic design.

This chapter should give the reader a basic understanding of the design of the XNU kernel, as well as some more in depth insights in the way XNU handles memory management and syscalls. This chapter details how the different components of XNU interact with each other, which is a necessary precondition when doing security research in XNU. The rest of this thesis assumes that the technical details presented in this chapter are known to the reader.

2.2 The design of the XNU kernel

2.2.1 General overview

The XNU kernel is separated into three core components that heavily interact. Each component has its own origin and a specific task and reason to be included into the XNU kernel. The components are: the original Mach kernel, specific parts of the FreeBSD kernel and Apple's own I/O Kit framework for drivers.

At the basis of XNU lies the Mach kernel. The Mach kernel was developed by Carnegie Mellon University from 1985 until 1994. Mach is one of the few true micro kernels, which means that things like a network stack or a virtual file system are not included. To add those features, NeXT copied them from the BSD kernel. In a true micro kernel concept, such as Mach, these functions should run in user space, but in the XNU kernel the BSD component also runs in kernel space.

When Apple adopted the XNU kernel, the Mach kernel was upgraded from version 2.5 to 3.0. Apple also added the I/O Kit framework. I/O Kit enables hardware manufactures to easily develop drivers for the OS X platform. I/O Kit offers a stable kernel framework and base classes for a wide spectrum of different

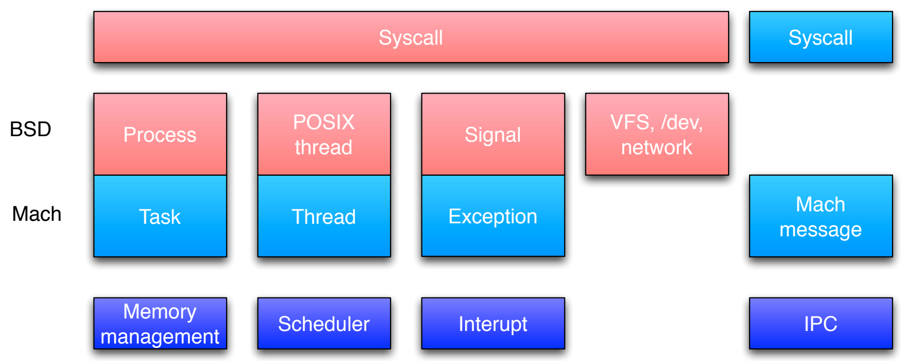


Figure 2.1: XNU design overview as sketched by Lucy[21]

hardware devices, such that hardware manufactures only have to implement a communication channel to the actual hardware.

In figure 2.1 one can see that BSD extends many of the Mach concepts. Where Mach introduces the concept of tasks, BSD extends this concept with its definition of processes, which adds things like a user ID and terminal to a task. Both Mach and BSD provide user space applications with various syscalls. BSD syscalls work by invoking the syscall handler with a syscall number, whereas Mach syscalls work by sending IPC messages to the kernel.

2.2.2 The different components

Mach

Being a micro kernel, Mach is responsible for all low level hardware interaction, the scheduler and the memory management. The Mach component can be found under `osfmk/` in the XNU source tree.

Mach is designed as a message passing service. Rather than invoking functions, Mach enables applications to send messages back and forth in order to communicate. The idea of message passing is widespread in the design and implementation of OS X and iOS.

BSD

The current BSD component is based on the FreeBSD kernel and can be found under `bsd/` in the XNU source tree. BSD is responsible for higher level constructs in the kernel, such as the network stack and the virtual file system. It also introduces the concept of user id's and UNIX file permissions. Beside these components it also extends many of the normal Mach concepts. It adds a BSD process to a Mach task, POSIX threads to Mach threads and signals where Mach uses exceptions.

As an example we look at the implementation of the Mach task struct, which has its BSD counterpart in a process struct. In listing 1 we can see that a Mach task has a pointer to its process struct and vice versa.

```
// osfmk/kern/task.h
struct task {
    ...
    void* bsd_info;
    ...
};

// bsd/sys/proc_internal.h
struct proc {
    ...
    void* task;
    ...
};
```

Listing 1: the `task` and `process` struct with a pointer to each other

I/O Kit

To reduce the complexity of driver development for XNU, Apple added a driver framework called I/O Kit. I/O Kit uses a minimal version of C++, called Embedded C++, which misses features that are, accordingly to Apple, unsafe for usage in a kernel, like multiple inheritance, templates and exceptions[4].

I/O Kit contains base classes for many different hardware types. Meaning that hardware vendors can focus on implementing hardware specific features, instead of writing an entire device driver from scratch. Base classes are called families in I/O Kit, for example there is a `com.apple.iokit.IOUSBFamily` family for USB and a family for human input devices called `com.apple.iokit.IOHIDFamily`. The I/O Kit source can been found under `iokit/` in the XNU source tree.

2.3 Syscalls

2.3.1 Syscalls in XNU

Syscalls are the main entry point to kernel space for user space applications. Syscalls cover a large portion of the kernel source and, because they can be called from user space, are a large attack vector. In this section we describe the calling mechanism for syscalls in the XNU kernel.

The XNU kernel has two separated systems for calling kernel functions, one for calling the BSD syscalls and one for the Mach syscalls (which are called Mach Traps). Each syscall is assigned a unique number, which is used by user space application by invoking the syscall handler with the number of the requested syscall. The BSD syscalls have positive syscall numbers to stay POSIX compliant and are equal to that of FreeBSD. The Mach syscalls have a negative number so that they don't interfere with the BSD syscalls.

Calling a syscall on OS X follows the System V ABI[23]. A syscall is invoked by the `syscall`[16] instruction. The syscall number is passed in the `rax` register, which will also hold the return value upon return. Syscalls normally support up to six arguments which are passed through using the `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9` registers respectably. If a syscall requires more than six arguments it will copy the arguments from the user space stack, pointed to by the `rsp` register, using the `copyin()` method (described in 2.3.4). As an example, the assembly code below will call `exit(42)` under OS X:

```
mov    rax, 0x2000001 ; exit has syscall number 1
mov    rdi, 0x2a      ; give 42 as argument
syscall               ; invoke the syscall handler
```

Listing 2: calling `exit()` on OS X

On iOS the calling convention is similar, but adapted for the ARM architecture. The arguments for syscalls are passed through the `R0`, `R1`, `R2`, and `R3` registers as defined in the ARM ABI[6]. When requiring more arguments the data is copied in from the user space stack. The syscall number is passed in the `R12` register, which is also called the Intra-Procedure-call scratch register or `IP` register for short (not to be mistaken with the Instruction Pointer, which is stored in `R15`). The syscall is invoked using the `svc 0x80`[7] call, which raises a software interrupt. As an example, the assembly code below will call `exit(42)` under iOS:

```
mov    r12, 0x1 ; exit has syscall number 1
mov    r0, 0x2a ; give 42 as argument
svc    0x80     ; invoke the syscall handler
```

Listing 3: calling `exit()` on iOS

2.3.2 BSD syscalls

All BSD syscalls ends up in the so called `sysent` table, defined in `bsd/kern/init_sysent.c` (the file can be generated by running `bsd/kern/makesyscalls.sh syscalls.master`). The `sysent` table holds a `sysent` struct for each syscall, which is shown in listing 4.

The BSD syscalls are handled by the `unix_syscall64()` method for 64-bits applications and the `unix_syscall()` method for 32-bits applications. Both are defined in `bsd/dev/i386/systemcalls.c`. The methods have one argument: a pointer to a so called saved state struct. The saved state struct holds the value of all CPU registers at the moment of kernel entry. The `unix_syscal[64]()` methods obtains the appropriate sysent struct based on the syscall number, they will then check if the number of arguments is higher than the maximum amount that can be passed through the CPU registers. If the number of arguments is

```
bsd/sys/sysent.h
struct sysent {     /* system call table */
    int16_t     sy_narg;    // number of args
    int8_t      sy_resv;    // reserved
    int8_t      sy_flags;   // flags
    sy_call_t   *sy_call;   // implementing function
    sy_munge_t  *sy_arg_munge32; // system call arguments
                                 // munger for 32-bit process
    sy_munge_t  *sy_arg_munge64; // system call arguments
                                 // munger for 64-bit process
    int32_t     sy_return_type; // system call return types
    uint16_t    sy_arg_bytes;   // Total size of arguments in
                                // bytes for 32-bit syscalls
};
```

Listing 4: the `sysent` struct

too high they will invoke the `copyin()` method to copy the arguments to kernel space. In the end they will call the syscall by the `sy_call` function pointer in the `sysent` struct.

2.3.3 Mach Traps

The Mach syscalls are called through the IPC subsystem, but behind the scenes it uses the syscall mechanism. The various Mach Traps are combined in a large table called the `mach_trap_table`, similar to the BSD `sysent` table. There are currently a little over 30 active traps, much less than the number of BSD syscalls (more than 300). Each trap in the `mach_trap_table` has a `mach_trap_t` structure, which is shown in listing 5.

```
osfmk/kern/syscall_sw.h
typedef struct {
    int          mach_trap_arg_count;
    int          (*mach_trap_function)(void);
    ...
    #if MACH_ASSERT
    const char*  mach_trap_name;
    #endif /* MACH_ASSERT */
} mach_trap_t;
```

Listing 5: the `mach_trap_t` structure

The `mach_trap_table` is defined in `osfmk/kern/syscall_sw.c`. A trap is invoked by the `kernel_trap()` macro, which is machine dependent (the x86_64 version is defined in `osfmk/mach/i386/syscall_sw.h`). The `kernel_trap()` macro will lookup the syscall number in the `mach_trap_table` and call the appropriate method.

2.3.4 Moving data from user to kernel space

If a syscall has to many arguments, or the arguments cannot be stored in a CPU register (like a string for instance) the data must be copied from user space into kernel space. In XNU the `copyin()` method is used to copy data from user space to kernel space, whereas the `copyout()` method works the other way around by coping data from kernel space to user space. Both the `copyin()` and the `copyout()` method are wrappers around the `copyio()` method, which is defined in `osfmk/x86_64/copyio.c` (for the x86_64 architecture).

In the old 32-bit version of XNU, used in Mac OS X until Snow Leopard, XNU did not share its address space with user space. Instead the kernel had its own virtual address space, requiring a context switch on every call to the kernel. Nowadays the XNU kernel shares its address space with the user space application. This allows a much simpler implementation of the `copyin()` function (before a special copy window had to be created with a shared memory segment). The `copyio()` function will do some initial checking before invoking the `_bcopy()` function, which is defined in `osfmk/x86_64/locore.s` for the x86_64 architecture. The `_bcopy()` function will copy the data directly from user space based on a source and destination address and a length parameter.

2.4 Memory management

2.4.1 Virtual memory

In modern operating systems memory is virtual, just like the file system. For each process it will look like he owns the entire address space. The memory manager is responsible for mapping virtual memory to physical memory. Under XNU a virtual memory map is represented by a `_vm_map` struct, defined in `osfmk/vm/vm_map.h`. Because not the entire virtual memory address space is mapped at any given moment, the virtual memory map is divided in several entries, each representing a continuous block of mapped memory which share common properties. Each entry is represented by a `vm_map_entry` struct (also defined in `osfmk/vm/vm_map.h`), which holds access rights information for example. Figure 2.2 illustrates the concepts of the virtual memory map.

Entries are added to the map by the `vm_map_enter()` method (`osfmk/vm/vm_map.c`). An entry can span multiple pages, but always has a single backing store (such as physical memory or a hard drive). The backing store is represented by a `vm_object` struct (`osfmk/vm/vm_object.h`), which is included in every entry.

The virtual memory map is used by all other memory allocators in the kernel, as well as in all user space applications. All allocators described next work on virtual memory.

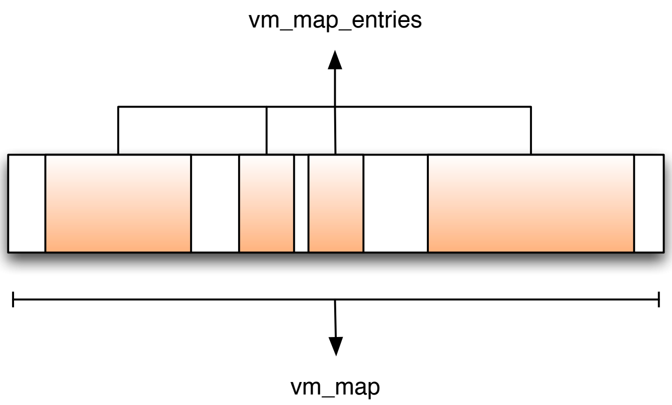


Figure 2.2: the vm_map with multiple entries

2.4.2 Process stack

Each user space thread has a corresponding stack in kernel space, which is used whenever code gets executed in kernel space (when calling a syscall for example). The kernel stack base pointer is declared in the `thread` struct, which is defined in `osfmk/kern/thread.h`. The kernel stack works just like its user space equivalence and follows the platforms ABI convention.

Whenever a thread gets started, the `kernel_thread_create()` function will invoke `stack_alloc()`, which is defined in `osfmk/kern/stack.c`. The `stack_alloc()` method will allocate a new stack and attach it to the specified thread. Stacks are stored in a stack free list, which is a single linked list. If the list is empty a new memory page is requested which will form the new stack for the thread. This means that kernel stacks are not necessarily adjacent in XNU.

2.4.3 The heap allocators

Allocating kernel memory on the heap is required when data should be preserved even when the current method returns and the current stack frame is removed from the stack. For this the XNU kernel provides multiple routines to choose from, depending on the location and requirements of the calling method. There are routines that will return aligned or wired memory for example, or routines that should only be called from the BSD or I/O Kit sections of the kernel. Some methods store meta information in- or outbound, others leave bookkeeping to the calling method. This section will give an overview of most common routines for allocating memory in the kernel. Figure 2.3 shows the most common path for allocating kernel memory.

In practice there are two allocators being used: the zone allocator (invoked by calling `zalloc()`) and the `kmem_alloc()` routine. The zone allocator is used for allocations up to 8 KB, whereas for larger allocations the `kmem_alloc()` routine is used. The `kmem_alloc()` routine will return a newly mapped page,

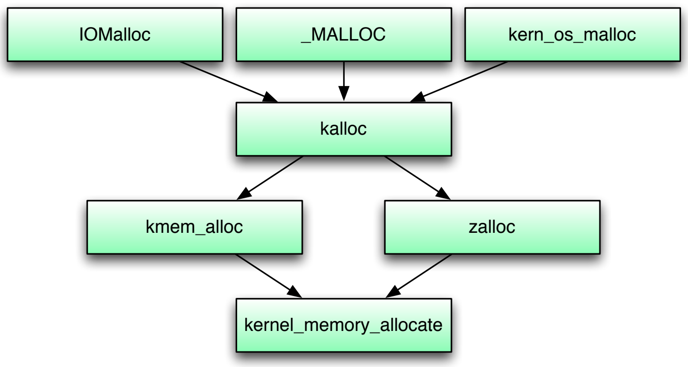

$$\begin{tikzcd}
\text{IOMalloc} \arrow[dr] & \text{\_MALLOC} \arrow[d] & \text{kern\_os\_malloc} \arrow[dl] \\
& \text{kalloc} \arrow[dl] \arrow[dr] & \\
\text{kmem\_alloc} \arrow[dr] & & \text{zalloc} \arrow[dl] \\
& \text{kernel\_memory\_allocate} &
\end{tikzcd}$$

Figure 2.3: overview of heap allocating functions

whereas `zalloc()` manages its own pages and keeps track of free blocks. Because allocations rarely exceed the 8 KB mark, most allocations end up in the zone allocator.

The allocation routines, except for the zone allocator, require no further explanation. Reading the source when needed should provide the reader with all necessary information. The zone allocator is a notable exception because of its size and frequent usage. The rest of this section will therefore focus on the zone allocator.

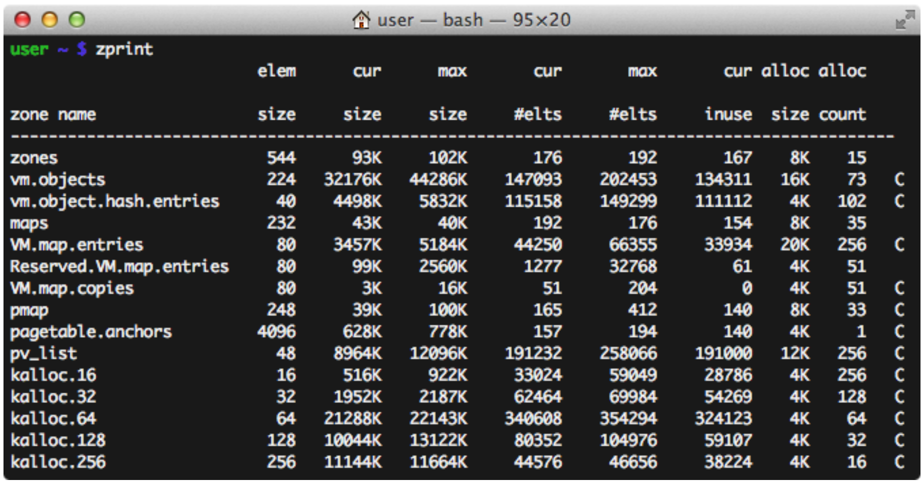
The zone allocator (which is defined in `osfmk/kern/zalloc.c`), divides its memory in zones. A zone has one or more associate pages which are divided into multiple blocks of the same size. Each zone has a name and is represented in the `zone` struct which is defined in `osfmk/kern/zalloc.h`. The current list of defined zones can be printed with the `zprint` command, which is shown in 2.4. A calling function will request a free block from a specified zone.

A zone keeps track of its free slots by an inbound single linked list. Each free block holds a pointer to the next free block, as can be seen in figure 2.5. When a new block is requested the allocator will pop the head of the free list for allocation. If a block gets freed its free pointer will point to the current head of the free list and the freed block will become the new head of the free list. The free list works on a ”first in, last out” bases.

The different zones are maintained by `zalloc()`. For the allocation of new memory it will invoke the `kernel_memory_allocate()` function. At boot time the kernel will create a list of default zones which are used by the `kalloc()` routine, called `kalloc.*` for all powers of two starting at 16.

2.5 Conclusion

The XNU kernel is special in a way that it consists of multiple components, each with their own background and design philosophy. Despite their differences, they are adapted in such a way they can communicate in order to make one



```
user ~ $ zprint
                          elem      cur      max       cur       max        cur alloc alloc

zone name                 size     size     size     #elts     #elts      inuse  size count
-------------------------------------------------------------------------------------------
zones                      544      93K     102K       176       192        167    8K    15
vm.objects                 224   32176K   44286K    147093    202453     134311   16K    73    C
vm.object.hash.entries      40    4498K    5832K    115158    149299     111112    4K   102    C
maps                       232      43K      40K       192       176        154    8K    35
VM.map.entries              80    3457K    5184K     44250     66355      33934   20K   256    C
Reserved.VM.map.entries     80      99K    2560K      1277     32768         61    4K    51
VM.map.copies               80       3K      16K        51       204          0    4K    51    C
pmap                       248      39K     100K       165       412        140    8K    33    C
pagetable.anchors         4096     628K     778K       157       194        140    4K     1    C
pv_list                     48    8964K   12096K    191232    258066     191000   12K   256    C
kalloc.16                   16     516K     922K     33024     59049      28786    4K   256    C
kalloc.32                   32    1952K    2187K     62464     69984      54269    4K   128    C
kalloc.64                   64   21288K   22143K    340608    354294     324123    4K    64    C
kalloc.128                 128   10044K   13122K     80352    104976      59107    4K    32    C
kalloc.256                 256   11144K   11664K     44576     46656      38224    4K    16    C
```

Figure 2.4: overview of defined zones

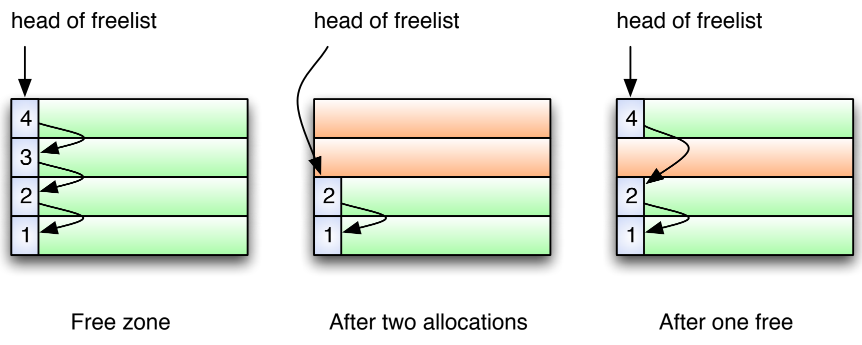


Figure 2.5: example of zone allocations

working kernel. Each component has a specific list of tasks within the the overall design of the kernel.

This concludes the introduction material for this thesis. The next part will describe all security mechanisms present in todays version of XNU. We start this part with an overview of memory protection mechanisms in the kernel. Because most kernel vulnerabilities will trigger some sort of memory overwrite, memory protection is of great importance.

Part II

XNU security features

3 Memory protection

3.1 Introduction

Memory protection works on the premises that applications (and thus the kernel) will always contain vulnerabilities that will allow an attacker to alter memory of some sort. Rather than trying to find and fix the vulnerabilities itself, nowadays protection mechanisms try to limit the damage that can be done. Most kernel vulnerabilities concern some sort of memory overwrite. By carefully corrupting memory an attacker could gain code execution in kernel space. There are many different types of memory corruption vulnerabilities, like for example a stack based buffer overflow, double free or write anywhere. Therefore, a single mitigation technique that can withstand all different attacks cannot exist. Modern operating systems implement different techniques that combined should prevent memory corruption vulnerabilities from becoming exploitable.

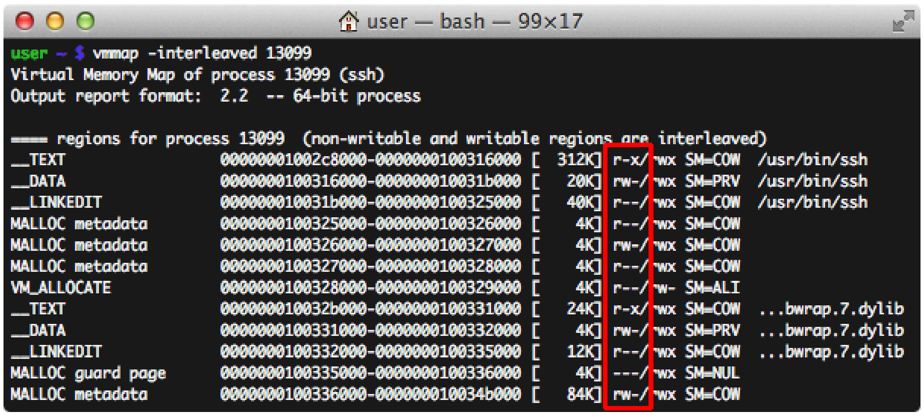
User space applications nowadays come with a whole arsenal of techniques which try to protect them against attackers. This has resulted in vulnerabilities that require either a higher skilled attacker or even have become unexploitable. After user space, manufacturers started to implement the same mechanisms in kernel space. Not all mechanisms have been implemented with the same level of effectiveness, due to the strict performance rules that apply in kernel space. A kernel that operates a few percent slower due to security checks has its effect on the performance of the entire operating system. Therefore protection mechanisms in kernel space must be low in required resources.

Apple only recently started with the implementation of protection mechanisms in kernel space. OS X Mountain Lion and iOS 6 contain new mechanisms (such as kernel ALSR) and others have been greatly improved (such as WˆX). This shows that Apple is actively trying to harden its kernel against attackers. In this chapter we describe the techniques present in todays release.

3.2 WˆX protection

WˆX, which stands for Write XOR eXecute, is a technique which marks a memory page either writable or executable, but never both. WˆX, sometimes referred to as NX (Never eXecute), XD (eXecute Disable) or DEP (Data Execution Prevention), is already widespread in user space applications. It was first implemented on OpenBSD[13] and currently supported by hardware and all other mainstream operating systems. WˆX protection is twofold. It prevents the patching of code in memory, by marking the code pages executable but not writable. It also protects against the execution of code from user controlled memory (such as the stack or the heap), a common used trick where the attacker would use a vulnerability to set the program counter to user controlled memory, to gain arbitrary code execution. W^X prevents this attack by marking the pages that hold user controlled information as non executable.

For user space applications one can use the `vmmap` command to print the currently mapped memory pages and their permissions. Figure 3.1 shows some of the output of `vmmap` of a running process. The red bordered column shows the current page permissions. In this example the first `_TEXT` section (holding program code) is executable, but not writable and the first `_DATA` section (holding initialized global and local static variables) is writable, but not executable.


```
user ~ $ vmmap -interleaved 13099
Virtual Memory Map of process 13099 (ssh)
Output report format:  2.2  -- 64-bit process

==== regions for process 13099  (non-writable and writable regions are interleaved)
__TEXT                 00000001002c8000-0000000100316000 [  312K] r-x/rwx SM=COW  /usr/bin/ssh
__DATA                 0000000100316000-000000010031b000 [   20K] rw-/rwx SM=PRV  /usr/bin/ssh
__LINKEDIT             000000010031b000-0000000100325000 [   40K] r--/rwx SM=COW  /usr/bin/ssh
MALLOC metadata        0000000100325000-0000000100326000 [    4K] r--/rwx SM=COW
MALLOC metadata        0000000100326000-0000000100327000 [    4K] rw-/rwx SM=COW
MALLOC metadata        0000000100327000-0000000100328000 [    4K] r--/rwx SM=COW
VM_ALLOCATE            0000000100328000-0000000100329000 [    4K] r--/rw- SM=ALI
__TEXT                 000000010032b000-0000000100331000 [   24K] r-x/rwx SM=COW  ...bwrap.7.dylib
__DATA                 0000000100331000-0000000100332000 [    4K] rw-/rwx SM=PRV  ...bwrap.7.dylib
__LINKEDIT             0000000100332000-0000000100335000 [   12K] r--/rwx SM=COW  ...bwrap.7.dylib
MALLOC guard page      0000000100335000-0000000100336000 [    4K] ---/rwx SM=NUL
MALLOC metadata        0000000100336000-000000010034b000 [   84K] rw-/rwx SM=COW
```

Figure 3.1: `vmmap` output of a SSH process

In OS X Mountain Lion and iOS 6, Apple improved the number of pages that are W^X protected in the kernel. The stack and the heap are marked non-executable, as it can contain user controlled data. This prevents the execution of code from user controlled data, meaning the attacker has to find new ways of executing code in kernel space.

Besides the non-executable heap and stack the code section can no longer be altered. Altering the code pages is used by the jailbreaking community to patch certain routings in kernel space. Most notably the disabling of the code signing enforcement, so that iOS will run custom signed binaries.

Important kernel structs such as the `sysent` struct (see Chapter 2 for more information about the `sysent` struct) are placed in read-only memory. It was a common trick used to inject a new syscall into the `syscall` table, and then call this syscall from user space. This gives the attacker control over the instruction pointer.

Because of the more strict W^X implementation, the old kernel exploitation techniques, where the attacker would run his shellcode from stack or heap memory, no longer work. The patching of program code or in memory stored structs is also mitigated. Attackers now need to find ways to disable the W^X protection or to find a way around it, for example by chaining existing code chucks. A technique commonly referred to as Return Oriented Programming (ROP)[18].

3.3 NULL pointer dereference protection

```
struct ucred* test = 0;
test->cr_uid = 0;

int (*ptr)() = 0;
ptr();
```

Listing 6: NULL pointer dereference

Consider the code of listing 6 in kernel space. Here two pointers are set to zero after which they are either dereferenced or called. Since there is nothing mapped at this location, the kernel will crash. The attacker can use this to its advantage by mapping a memory page at address 0 (using the `mmap()` syscall). A precondition for this attack is that the kernels shares its address space with user space and that user space has control over the first page of virtual memory. In case of a function pointer the kernel will execute the attackers payload with kernel privileges. When the pointer points to a variable or struct, the attacker can try to change the program flow by altering the values to its advantage. Such a vulnerability is referred to as a NULL pointer dereference vulnerability.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>

int main(int argc, char **argv) {
    void* page = mmap((void*)0x0, 4096, PROT_READ
            | PROT_WRITE | PROT_EXEC, MAP_PRIVATE
            | MAP_ANON | MAP_FIXED, -1, 0);

    if ((long)page < 0) {
        perror("mmap");
    } else {
        printf("page mapped at %p\n", (void*)page);
    }

    return 0;
}
```

Listing 7: mapping a memory page at address 0

The exploitability of such vulnerability depends on whether the attack can map memory on address 0. Listing 7 shows a piece of code which tries to allocate a memory page at address 0. The success depends on the architecture. On OS X this code will succeed if it runs in 32 bit mode, for this the application needs to be compiled as a 32-bits executable (`llvm-gcc -m32`). Since in 32-bit mode the kernel does not share its address space with user space there is no harm in allocating a memory region at address 0, thus `NULL` pointer deferences are not exploitable.

OS X applications that are compiled as 64-bits applications share their virtual address space with the kernel. The bottom 128 TB can be used by the application as the top 128 TB is marked as kernel memory. Trying to access memory above the 128 TB border would raise an access violation error. When loading the Mach-O binary a special memory page is added, the so called page zero. This page is not actually mapped in memory, but prohibits the mapping of another page at address 0. The page zero page is 4GB in size by default, meaning the first 4 GB of memory in a 64 bits application cannot be mapped by the application. This effectively protects the kernel against `NULL` pointer dereferences. The implementation of the page zero mapping in kernel can be seen in listing 8.

```
bsd/kern/mach_loader.c
static
load_return_t
load_segment(
    ...
)
{
    ...
    /*
     * This is a "page zero" segment:  it starts at address
     * 0, is not mapped from the binary file and is not
     * accessible. User-space should never be able to access
     * that memory, so make it completely off limits by
     * raising the VM map's minimum offset.
     */
    ret = vm_map_raise_min_offset(map, seg_size);
    if (ret != KERN_SUCCESS) {
        return (LOAD_FAILURE);
    }
    return (LOAD_SUCCESS);
    ...
}
```

Listing 8: part of the page zero implementation

As we will describe in Chapter 4, mapping a new memory page under iOS (either writable or executable) would violate the code signing restrictions and is therefore not possible. Furthermore mapping a memory page with both write and execute permissions is forbidden and enforced by the kernel. Mapping a page with such wide permissions will halt execution, leaving `NULL` pointer dereference bugs unexploitable on iOS.

To conclude, `NULL` pointer dereference vulnerabilities are not exploitable on OS X nor iOS. Either because mapping a page on address 0 is not possible or harmless.

3.4 Kernel ASLR

Address Space Layout Randomization (ASLR) is a familiar technique to prevent exploitation in user spaces, by loading memory pages on a random offset in memory for each new process. Which memory pages are randomized depends on the implementation. For iOS 6 and OS X Mountain Lion this means that the stack and heap are randomized each time an application starts. However the libraries are only randomized after each reboot, this is different from Linux for example where library randomization is done for each new process. The current ASLR implementation for user space is therefore easy to bypass for local attackers, as the attacker can determine the random offset (with `gdb` or `vmmap` for example) before starting his exploit.

In iOS 6 and OS X Mountain Lion Apple implemented ASLR in the kernel. Prior to those releases the kernel was mapped at a deterministic location. Kernel ASLR should prevent exploitation as for attackers precise knowledge of parts of the memory map is required. For kernel ASLR the kernel is compiled as a position independent executable, which can be seen by executing `otool -hv /mach_kernel`. Meaning that all calls, jumps and references are relative in the binary. No absolute addresses are used.

Randomization is done by the addition of two random offsets. The first random offset is generated by the boot loader and is used to slide the entire kernel image in memory. The second random offset is generated in the early boot process of the kernel and is used to slide the `kernel_map`, which holds the heap and stack. Two random offsets are used so that if the attacker can leak a stack or heap address, he or she gains no extra information on where the code segment lies in memory.

The boot loader (iBoot for iOS and `boot.efi` for OS X) will generate a random number and pass this to the kernel in the `boot-args` struct. For OS X the random number is generated by using the `rdrand` instruction[34], on iOS the current tick counter is used. The random number is then used in the formula `0x010000000 + (random_number * 0x00200000)`, giving the slide offset. The kernel base will slide into higher addresses using the slide offset. The slide offset has an entropy of 8 bits, with gaps of 2 megabyte.

Apart from the kernel base offset an extra random value is generated to which the `kernel_map` will slide in memory. The `kernel_map` will hold the heap and stack during execution. For the `kernel_map` a 9 bits random number is generated in `vm_mem_bootstrap()` (`osfmk/vm/vm_init.c`), which is multiplied by the page size (shown in listing 9). This gives 512 possible addresses in a 2 MB range.

ASLR is only effective if the attacker is unable to obtain any information about the memory map. Prior to the release of iOS 6 and Mountain Lion many syscalls returned kernel addresses as part of their specification. An example is the `copyLoadedKextInfo()` call, which is used by the `kextstat` utility. It will osfmk/vm/vm_init.c

```
/*
 * Eat a random amount of kernel_map to fuzz subsequent heap,
 * zone and stack addresses. (With a 4K page and 9 bits of
 * randomness, this eats at most 2M of VA from the map.)
 */
if (!PE_parse_boot_argn("kmapoff", &kmapoff_pgcnt,
    sizeof (kmapoff_pgcnt)))
    kmapoff_pgcnt = early_random() & 0x1ff; /* 9 bits */

      if (kmapoff_pgcnt > 0 &&
  vm_allocate(kernel_map, &kmapoff_kaddr,
  kmapoff_pgcnt * PAGE_SIZE_64, VM_FLAGS_ANYWHERE) !=
  KERN_SUCCESS)
    panic("cannot vm_allocate %u kernel_map pages",
        kmapoff_pgcnt);
```

Listing 9: calculating the `kernel_map` randomization

return a list with information about the loaded kernel extensions, including the loading address. Apple tried to obfuscate all kernel pointers that leak to user space, by unsliding them, picking a random offset or by just setting the pointer to 0. An example is shown in listing 10, where the kernel extension load address is being unslide. As a result the `kextstat` utility shows invalid kernel addresses on OS X Mountain Lion.

libkern/c++/OSKext.cpp

```
// osfmk/mach/vm_param.h
#define VM_KERNEL_UNSLIDE(_v)                         \
        ((VM_KERNEL_IS_SLID(_v) ||                    \
          VM_KERNEL_IS_KEXT(_v)) ?                    \
            (vm_offset_t)(_v) - vm_kernel_slide :     \
            (vm_offset_t)(_v))

OSDictionary *
    OSKext::copyInfo(OSArray * infoKeys)
{
    ...
    loadAddress = VM_KERNEL_UNSLIDE(loadAddress);
    ...
}
```

Listing 10: unsliding the load address

3.5 Heap protection

Corrupting the heap's internal state is a common attack technique for gaining code execution, for example the kernel exploit used to jailbreak iOS 5.0.1[27] was a heap based overflow in mounting a corrupted HFS+ image. But also in user space heap based vulnerabilities are common[37][2]. Due to the rise of stack protection mechanisms, heap exploits gain popularity.

We can classify different heap based vulnerabilities such as a double free, where a block of heap memory is freed twice or a use-after-free where a memory block is used after is was already freed. Heap exploits tend to overwrite either the contents of an allocated block or some inbound meta information. Heap allocators often store heap state information inbound, which also is the case with most heap allocators in the XNU kernel (which can be seen in Chapter 2.4). Corrupting the inbound meta data is a common technique which could lead to code execution[10].

Most user space heap allocators try to protect themselves against attackers by checking the heap state before each (de)allocation[29]. In the XNU kernel the `_MALLOC()`, `kern_os_malloc()` and `zalloc()` all store meta information inbound. The integrity of the inbound meta data gets compromised if the attacker is able to overflow an allocation, or access an already freed block. Due to the fact that performance is of such importance in kernel space, only basic checking is done at this moment. Most checks are not meant for withstanding attacks, but merely try to detect bugs in an early stage before harm can be done.

The `kalloc()` function will keep track of its largest allocated block by storing its size in `kalloc_largest_allocated`. When freeing a memory block the size is compared to that of the largest allocated block at the time and halts execution if the size is larger, which offers some protections against unwanted frees. If an exceptionally large size would be freed, blocks that are still in use would also be return to the system. This would leave the system in an unpredictable state. However scenarios where the freed length is invalid but smaller than the largest allocation go unnoticed, which would still result in unwanted frees.

The zone allocator has an inbound free list which could give an attacker control over where newly allocated blocks are allocated in memory. For OS X the zone leak detector is enabled, which tries to detect overflows in a basic manner. On deallocation a block will be marked either poisoned or non poisoned (currently every 16th deallocation will be poisoned), by placing a marker behind the next pointer. A poisoned block has the magic marker `0xfeedface` and a non poisoned block uses `0xbaddecaf`. If the block is poisoned its content will also be overwritten with the magic marker `0xdeadbeef` and a clean copy of the next pointer will be added at the end of the block. The difference between a poisoned and non poisoned block is shown in figure 3.2.

On allocation it will be determined if the block was poisoned, and if so the content will be checked for containing only the `0xdeadbeef` value. The next pointer is also compared to the one stored at the end of the block. This is done to detect small overflows. Suppose the attacker is able to overflow into the adjacent heap block, which is unallocated. The first few bytes will contain the

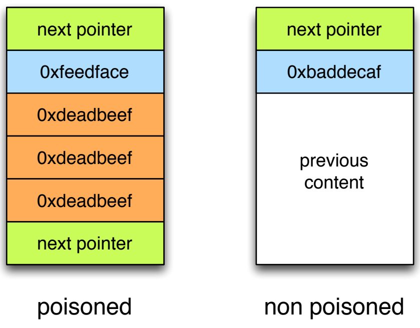


Figure 3.2: the contents of a free poisoned and non poisoned zone block

next pointer. Without the next pointer at the end the attacker would be able to overflow only the next pointer, leaving the rest of the block untouched.

The leak protector is implemented by `alloc_from_zone()` and `free_to_zone()`, both inline functions defined in `osfmk/kern/zalloc.c`.

3.6 Stack canaries

Cowan et al.[11] presented a way to protect the stack as exploitation target, which led to multiple implementations in compilers, such as GCC's Stack-Smashing Protector (SSP) and Microsoft's /GS.

Stack protection uses a canary value (or sometimes called stack cookie) to detect stack based buffer overflows. A stack canary is a random value that is placed on the stack before the stored CPU registers, as shown in figure 3.3.

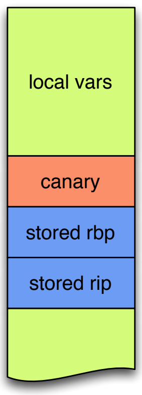


Figure 3.3: canary value on the stack

Before returning to the calling function, the canary value on the stack is compared to a known good backup. If the two do not match, the canary value on the stack has been altered and execution is halted as can been seen in listing 11. Stack canaries protect the stack against scenarios where the attacker is able to overflow any of the local variables. Without stack canaries the attacker can overwrite the stored instruction pointer, giving him control over the execution flow.

```
mov  rax, qword [ds:rax]     ; load known good backup
mov  rdx, qword [ss:rbp-8]   ; load stack canary
cmp  rax, rdx                ; compare the two
mov  qword [ss:rbp-16], rcx  ; if the two are not equal,
jne  0xffffff8000269929      ; jump to -
mov  rax, qword [ss:rbp-16]  ;           |
add  rsp, 0x10               ;           |
pop  rbp                     ;           |
ret                          ;           |
call  ___stack_chk_fail      ; <--------
nop                          ; ___stack_chk_fail will
endp                         ; cause a kernel panic
```

Listing 11: checking of the stack canary at the end of a function

During boot the canary value is randomly generated in `osfmk/x86_64/start.s` and the second byte is set to zero as seen in listing 12. This is done to break the various string functions, as C-style strings are 0 terminated. Meaning that the stored CPU registers can never be overwritten by the attacker. Even if he or she would be able to determined the canary value in advance.

osfmk/x86_64/start.s

```
Lstore_random_guard:
    xor %ah, %ah  //Security: zero second byte of
                  // stack canary
    movq    %rax, ___stack_chk_guard(%rip)
```

Listing 12: zero the second byte of the stack canary

3.7 Conclusion

XNU is feature rich when it comes to memory protection mechanisms All techniques combined raise the bar for attackers to develop a reliable exploit for a vulnerability in XNU. Attackers now might need to chain multiple vulnerabilities in order to bypass all memory protection mechanisms.

Memory protection is not the only way the XNU kernel tries to protect the integrity of the system. Under iOS and OS X Mountain Lion code signing and sandboxing try to limit the attack surface by only loading trusted applications and limit their resources. Code signing and sandboxing are discussed in the next chapter.

4

Code signing and sandboxing

4.1 Introduction

Code signing is an important security concept in iOS, which now also has been ported to OS X Mountain Lion. By adding a digital signature to applications their integrity can be assured and the identity of the developer can be verified. This should reduce the risk of malware infections, since malware would not pass Apple's strict App policy and will, as a result, lack the required signature. This design only works if code signing is actually enforced by the operating system. For iOS this enforcement is enabled by default without a way to disable this for the end user. OS X Mountain Lion introduces Gatekeeper which brings the concept of code signing to the Mac, but can, at least for now, be disabled by the user.

Code signing is implemented by adding a signature for each segment in a Mach-O binary. The loader of the binary will check the signature for each segment before loading it into memory. The application may only run if it has valid signatures for each segment. For checking the signatures the loader has a copy of Apple's public key.

Besides code signing, applications also run inside a sandbox, restricting file, network and syscalls access. For example the PDF renderer in Preview.app does not allow file access. This limits the possibilities of an attacker whenever he gains code execution by abusing a vulnerability in the PDF render engine.

This chapter will look into the implementation of code signing and sandboxing and how this is used to protect the kernel. We start with the description of code signing. Since there are differences between code signing in iOS and OS X this section will describe both implementations. In the second part of this chapter the sandbox implementation will be discussed. In the conclusion we discuss the impact these features have on kernel exploitation.

4.2 Code signing

4.2.1 The iOS secure boot chain

Apple tries its best to ensure the integrity of the iOS operating system. This integrity checking is done by code signing and starts as soon as the device is turned on. The entire process of booting an iOS device is subjected to code signing such that only firmware images from Apple can be booted. This process is described below.

Boot ROM

The Boot ROM is the first stage in booting an iOS device. The Boot ROM is stored on a piece of read-only memory, which is embedded in the system chip (like the A5 chip which is used in the iPhone 4S and iPad 2). Programming the Boot ROM in read-only flash memory eliminates the risk of tampering, and ensured its integrity. The Boot ROM contains Apple's root CA, which is used for signature verification during the rest of the boot chain. The Boot ROM is also handles the DFU mode (device firmware upgrade) for flashing new firmware images on the device.

The Boot ROM will decrypt the Low-Level Boot loader (LLB), which is encrypted with the so called GID key. The GID key is embedded in the hardware and equal for all devices with the same type of system chip. It cannot be extracted via software. After decryption the Boot ROM will verify the signature of the LLB before loading it into memory and handing over the execution. The LLB is the first application that is loaded from writable flash memory and is therefore susceptible to tampering. Code signing prevents tampering by halting execution when a invalid signed binary is provided.

Since the Boot ROM is embedded into the hardware of the device, Boot ROM updates require a new hardware revision. This makes Boot ROM exploits valuable, since Apple is unable to provide a fix for already released devices. There have been several Boot ROM exploits in the past, such as the 0x24000 segment overflow for the iPhone 3GS[1] and the Limera1n exploit for the A4 devices (iPhone 4, iPad 1, iPod 4G)[2], allowing to break the secure boot chain by booting an unsigned LLB.

Low-Level Boot loader

The Low-Level Boot loader is the first stage in the iOS boot process that uses code signing. Replacing the LLB with an unsigned patched version will fail to boot because the signature check done by the Boot ROM will fail. The LLB will do some initial setup and initialization after which it will read the iBoot boot loader from memory and decrypt the image using the GID key. It will check the signature of iBoot before handing over the boot process.

The LLB is stored in `Firmware/all_flash/all_flash.*.production/LLB.*.RELEASE.img3` inside the IPSW firmware file. Decryption keys can be found on the iPhone Wiki[3] for the appropriate firmware version.

[1]theiphonewiki.com/wiki/index.php?title=0x24000_Segment_Overflow
[2]theiphonewiki.com/wiki/index.php?title=Limera1n_Exploit
[3]theiphonewiki.com/wiki/index.php?title=Firmware

iBoot

iBoot is the second stage boot loader. It supports USB and serial for recovery purposes. During a normal boot process iBoot will load the kernel from memory, decrypt the image using the GID key and check its signature. If the signature is valid, iBoot will disable all access to the GID key until the next reboot before handing over the boot process to the kernel. Because of this restriction it is only possible to decrypt kernel image from devices that are vulnerable for a Boot ROM vulnerability, so that the decrypted kernel image can be dumped before the access is restricted. There are currently no known Boot ROM vulnerabilities for the newer devices (with an A5 or A6 chip).

The iBoot firmware image is stored in `Firmware/all_flash/all_flash.*.production/iBoot.*.RELEASE.img3` inside the IPSW firmware file. Decryption keys can be found on the iPhone Wiki for the appropriate firmware version.

XNU kernel

The kernel will mount the system partition as read-only file system on the `/` mount point. Making this partition read-only will make sure that all system files stay intact after a forced shutdown of the device. It will also mount a data partition with write privileges under the `/private/var` mount point. This partition is used for storing user applications and data. After loading the file system all system daemons will start. The kernel checks the signatures of each loaded binary. This ensures that no unsigned binaries are able to run on iOS. Mandatory code signing such as on iOS cannot be disabled from user space, but only by patching the kernel in memory. Since kernel extensions from the attacker will also be unsigned, this patching has to be done via a kernel exploit.

The entire boot chain is shown in figure 4.1. In section 4.2.2 we show how application code signing is enforced by the kernel.

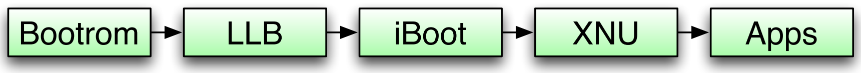
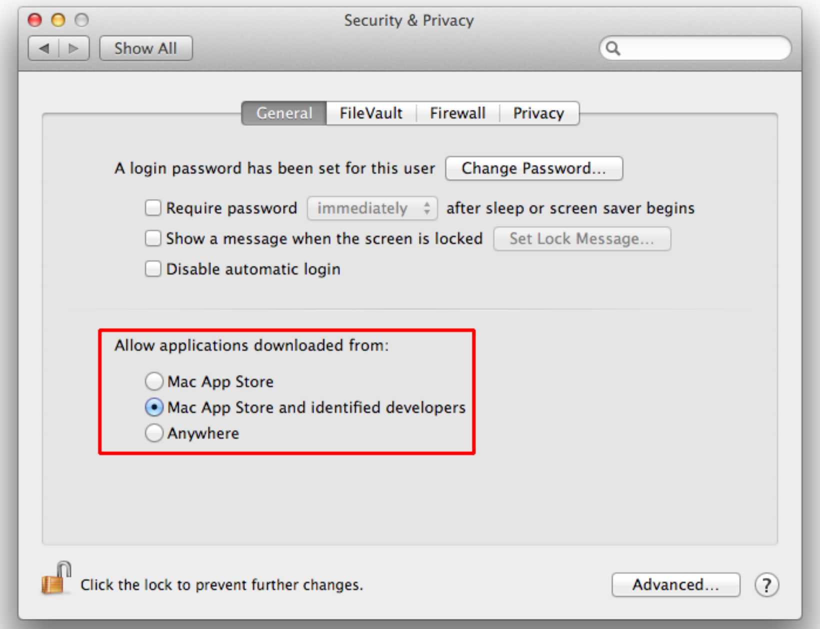


Figure 4.1: iOS boot process

4.2.2 Application code signing

Apple adopted the TrustedBSD project[38] in XNU, which implements a Mandatory Access Control (MAC) framework. The MAC framework allows custom policies to be registered. Both code signing and sandboxing are registered MAC policies in the XNU kernel.

For iOS all applications have to be signed since the launch of the App Store (iOS 2.0), this includes all system binaries that come with iOS. For the end user this code signing enforcement cannot be disabled, which makes that an end user can only install applications from the App Store.


Security & Privacy

General | FileVault | Firewall | Privacy

A login password has been set for this user [Change Password...]

☐ Require password [immediately] after sleep or screen saver begins

☐ Show a message when the screen is locked [Set Lock Message...]

☐ Disable automatic login

Allow applications downloaded from:

○ Mac App Store

[✓ Mac App Store and identified developers]

○ Anywhere

Click the lock to prevent further changes. [Advanced...]

Figure 4.2: the option menu for Gatekeeper

On OS X the user can optionally enable code signing, where it is called Gatekeeper. Enabling Gatekeeper should reduce the risk of malware infections, though its implementation is not as strict as on iOS. For example it is possible on OS X for a signed application to introduce new unsigned code by mapping a page with `rwx` permissions. Currently the user has three options: disable Gatekeeper, only allow signed applications and only allow signed application from the Mac App Store, as can be seen in figure 4.2.

osfmk/vm/vm_page.h

```
struct vm_page {
    ...
    cs_validated:1, // code-signing: page was checked
    cs_tainted:1,   // code-signing: page is tainted
    ...
}
```

Listing 13: code signing field in the `vm_page` struct

Code signatures are checked by `vm_fault_enter()` in the kernel, which is called every time a page fault occurs (including if the page is initially loaded into memory). The `vm_page` struct stores the current code signing state of the page, if it needs checking (tained) or has been validated (as can been seen in listing 13). The `vm_fault_enter()` function will request a check of the signature if needed. Signatures are stored inside the Mach-O binary, in the the `LC_CODE_SIGNATURE` section. They can be viewed using `otool`.

```
osfmk/vm/vm_page.h
kern_return_t
vm_map_enter(...) {
    ...
    #if CONFIG_EMBEDDED
        if (cur_protection & VM_PROT_WRITE){
            if ((cur_protection & VM_PROT_EXECUTE) &&    \
                    !entry_for_jit){
                printf("EMBEDDED: %s curprot cannot be  \
                    write+execute. turning off execute\n", \
                    __PRETTY_FUNCTION__);
                cur_protection &= ~VM_PROT_EXECUTE;
            }
        }
    #endif /* CONFIG_EMBEDDED */
    ...
}

kern_return_t
vm_map_protect(...) {
    ...
    #if CONFIG_EMBEDDED
            if (new_prot & VM_PROT_WRITE) {
                if ((new_prot & VM_PROT_EXECUTE) &&        \
                        !(current->used_for_jit)) {
                    printf("EMBEDDED: %s can't have both \
                        write and exec at the same time\n", \
                        __FUNCTION__);
                    new_prot &= ~VM_PROT_EXECUTE;
                }
            }
    #endif
    ...
}
```

Listing 14: enforcing the strict no `rwx` page policy

On iOS a memory page may also not be mapped with `rwx` permissions, which is enforced by both `vm_map_enter()` and `vm_map_protect()`. The `vm_map_enter()` function is called when the page is first allocated, `vm_map_protect()` when the page protection is changed. Having a `rwx` mapped page is forbidden as it would allow for new unsigned code to be introduced on the device. This rule was strictly enforced until iOS 4.3 where Apple chose performance over security with the introduction of a JIT compiler for Mobile Safari. The JIT compiler can compile JavaScript to native byte code, allowing faster execution of JavaScript. JIT implies introducing new code in memory, which breaks the mandatory code signing requirement, therefore Apple restrictes the use of JIT to Mobile Safari. Mobile Safari is the only process that is allowed to have a single `rwx` page in memory, trying to map a second `rwx` page in Mobile Safari will fail. Page permissions enforcing is currently only enabled on iOS, and not on OS X. The implementation can been seen in listing 14.

4.3 Sandboxing

Each application installed from the (Mac) App Store comes with a sandbox profile, designed to protect the system after exploitation. The sandbox not only limits file access and network capabilities, but also limits the syscalls a process is allowed to call. On iOS and on OS X with Gatekeeper enabled an attacker cannot run its own code due to the code signing restrictions, and thus the attacker can only achieve arbitrary code execution by exploiting an already signed application, meaning that the attacker is bound to the sandbox profile of the exploited process.

Just as code signing, the sandbox is implemented as a TrustedBSD policy in the closed sourced `com.apple.security.sandbox` kernel extension. The sandbox profiles are defined in the Scheme language and are compiled and delivered to the kernel by the `libsandbox.dylib` library, through the `mac_syscall` trap. An example sandbox profile is shown in listing 15.

The sandbox implementation is already covered by others[9][19], we will briefly discuss the implementation in kernel space. Most syscalls in the kernel have support for the sandbox policy. For example listing 16 shows the `connect()` syscall asking the TrustedBSD framework for permission.

The `mac_socket_check_connect()` function uses the `MAC_CHECK()` macro (defined in `security/mac_internal.h`), which iterates over all registered policies of the TrustedBSD framework, to ask for permission.

4.4 iOS jailbreaks

4.4.1 Tethered versus untethered jailbreaks

Jailbreaking is the process of removing the code signing enforcements from an iOS device. This allows for custom binaries to be installed on the device which have not been verified or approved by Apple. Because the kernel enforces the signing of binaries a kernel exploit is required in the jailbreaking process.

Because the entire boot process of iOS is also signed, the kernel image can not be overwritten with a version which has the code signing functionality removed. This patched kernel would not be boot by iBoot because of the invalid signature. This makes the difference between an tethered and untethered jailbreak. A tethered jailbreak means the jailbreak does not survive a reboot and needs to

```
/usr/share/sandbox/named.sb
;;
;; named - sandbox profile
;; Copyright (c) 2006-2007 Apple Inc.  All Rights reserved.
;;
;; WARNING: The sandbox rules in this file currently
;; constitute Apple System Private Interface and are subject
;; to change at any time and without notice. The contents of
;; this file are also auto-generated and not user editable;
;; it may be overwritten at any time.
;;
(version 1)
(debug deny)

(import "bsd.sb")

(deny default)
(allow process*)
(deny signal)
(allow sysctl-read)
(allow network*)

;; Allow named-specific files
(allow file-write* file-read-data file-read-metadata
  (regex "^(/private)?/var/run/named\\.pid$"
         "^/Library/Logs/named\\.log$"))

(allow file-read-data file-read-metadata
  (regex "^(/private)?/etc/rndc\\.key$"
         "^(/private)?/etc/resolv\\.conf$"
         "^(/private)?/etc/named\\.conf$"
         "^(/private)?/var/named/"))
```

Listing 15: an example sandbox profile

be jailbroken after each reboot. An untethered jailbreak is able to survive a reboot.

4.4.2 Tethered jailbreaks

Because of the code signing enforcement attackers cannot upload a kernel exploit to the device. The exploit would be unable to load. This means that attackers first need to take control over a validly signed application, from which they can stage their kernel exploit. The steps for a tethered jailbreaks are described below:

1. **Gain initial code execution**
   The first step in a jailbreak, gaining initial code execution in a user process.

bsd/kern/uipc_syscalls.c

```
int
connect_nocancel(__unused proc_t p, struct
connect_nocancel_args *uap,
__unused int32_t *retval)
{
    ...
    #if CONFIG_MACF_SOCKET_SUBSET
        if ((error = mac_socket_check_connect(
            kauth_cred_get(), so, sa)) != 0) {
            if (want_free)
                FREE(sa, M_SONAME);
            goto out;
        }
    #endif /* MAC_SOCKET_SUBSET */
    ...
out:
          file_drop(fd);
          return (error);
}
```

Listing 16: checking the sandbox policy

This can be done by exploiting a vulnerability in Apple's Mobile Safari browser, document viewer or any other service.

2. **Break out of the sandbox**
   The exploited process runs within a sandbox and unless the attacker has a kernel exploit that falls within the sandbox profile he or she first needs to break out of the sandbox. Because the attacker cannot introduce new code onto the device, this step has to be done using a 100% ROP payload. Unless the entry point in the previous step was Mobile Safari which has a JIT page which can be abused.

3. **Elevate privileges**
   Depending on the kernel vulnerability, it might be that the attacker needs to elevate privileges to those of the root user. If the kernel vulnerability can be triggered by the current user, this step can be omitted. If root privileges are required, a root process needs to be exploited.

4. **Run the kernel exploit**
   Only now the attacker has met all preconditions for his or her kernel exploit. The kernel exploit needs to be written by using ROP in the current process. The kernel exploit should allow the attacker to write in kernel memory.

5. **Patch kernel memory**
   After the kernel exploit, specific kernel routines have to be patched to disable the code signing restriction. Custom, unsigned, binaries can now be installed on the device. Because the kernel patches can only be done in memory, they will all vanish after a reboot.

4.4.3 Untethered jailbreaks

All kernel patches of a tethered jailbreak will be gone after a reboot, effectively meaning an iOS device always boots in a clean state. An untethered jailbreak should be able to make these patches persistent trough reboots. Because the kernel image cannot be altered due to the secure boot chain, an untethered jailbreak re-exploits the kernel after each reboot.

Untethered jailbreaks work by finding a vulnerability in one of the services that will be loaded at boot time (such as the VPN service). It should be possible to trigger this vulnerability by overwriting a configuration file, as they do not need a valid signature.

For example, for the jailbreak of iOS 5.1.1 a vulnerability in the mobile backup service was exploited that allowed the attackers to write outside the chrooted environment. This vulnerability was used to overwrite the default configuration file for the VPN service. Whenever the device was rebooted the VPN service would load the configuration file, containing an exploit for a vulnerability in the VPN service. After the attackers have control over the VPN service, a kernel exploit is staged. The kernel exploit allows the attackers to overwrite arbitrary kernel memory, which is used to patch the code signing routines.

4.5 Conclusion

The strict code signing enforcement on iOS means that kernel exploits has to be staged from a 100% ROP exploit in user space. This makes exploiting considerably more difficult as things like branching and looping are difficult to achieve using ROP. Under OS X, where code signing is less strict, this is generally less of an issue.

Sandboxing limits the attack surface, which might mean that a vulnerability cannot be triggered from within the application sandbox. The attacker is limited to the attack surface available from within the sandbox, or the attacker requires an extra vulnerability to establish a sandbox break.

Code signing and sandboxing are generally much harder to defeat than the memory protection mechanisms described in the previous chapter. Having a closed platform like iOS where only applications that have been reviewed and approved by Apple are allowed, require a much higher skilled attacker than on an open platform like OS X. The attackers require multiple exploits, for both user space and kernel space, in order to completely compromise a system.

The next part will cover steps an attacker can take to exploit the XNU kernel and how he or she might bypass the security mechanisms just presented. This part will start with an introduction on attack vectors in XNU and the techniques a attacker might use to find a vulnerability.

Part III

Exploitation techniques

5 Security research

5.1 Introduction

We have described the architecture of XNU and its protection mechanisms. In this chapter we focus on the threats of XNU and why attackers target the XNU kernel. There is a clear distinction between XNU exploitation on OS X and iOS. This will be described in the first section of this chapter.

The previous two chapters described the way XNU tries to protect itself and the rest of the system against attackers. The mechanisms do not close vulnerabilities, but raise the bar for exploitation. In the second part of this chapter we will discuss which parts of XNU are worth investigating and which approach should be used.

5.2 Strategy

5.2.1 OS X

An attacker would typically target OS X to gain root privileges. For this the attacker is bound to attack surface that is accessible for an unprivileged user. The XNU kernel is rarely targeted on OS X. Malware infections are slowly becoming more of a threat on OS X, but until now they run under a unprivileged user account. Or use traditional social engineering techniques, such as disguising themselves as a software updates, to obtain the system password[15].

Attackers are currently no real threat of the XNU kernel on the OS X operating system. This might also be because if the attacker gains access to a user's Mac, he already has full access to all its personal files, and thus has less need for root privileges.

5.2.2 iOS

The XNU kernel has existed for more than twenty years. Twenty years in which multiple people have done security research of some sort on XNU, either through fuzzing, source code auditing or any other technique. Almost all of that research was focused on attack surface that required no additional privileges, because the highest goal was, and for OS X still is, gaining a shell with root privileges.

This goal has changed since the introduction of code signing for iOS. Suddenly attackers require a kernel exploit for disabling code signing, so that custom binaries can be run. Often the attacker already has root privileges on the device, but cannot leverage from this because of the code signing enforcement. With the introduction of the strict code signing requirement, vulnerabilities that can only be triggered by the root user are suddenly valuable. If the attacker is targeting iOS he should focus on attack surface that is only accessible to the root user as this attack vector has been less researched.

An iOS attacker should take into account that code signing will limit the possibility of implementing complex kernel exploits, and that sandboxing might make certain attach surface unaccessible to the attacker.

Attackers on iOS are a threat to XNU. The so called jailbreaking is popular under iOS users, as it allows them to use applications that are unavailable on a non-jailbroken device. Looking at the list of public kernel exploits from the past few year we see that they are all found and used by jailbreakers.

5.3 Attack vectors

5.3.1 Syscalls

Syscalls are the most common way of interacting with the kernel. There are more than 300 BSD syscalls and another 45 Mach traps. Chapter 2 contains more information on syscalls.

Besides auditing rarely used syscalls, it is also interesting to see how syscalls interact with other syscalls, with respect to locking for example. The scheduler might choose to schedule out a syscall mid execution, which can lead to locking issues and/or race conditions. There are also syscalls that operate on base of a large switch block, such as `fcntl()` for example. Syscalls which contain a lot of code, specially ones with do a lot of branching, are interesting to audit. It is easy for a programmer to loose sight in all those code, which can lead to edge cases that have not been unaccounted for.

5.3.2 I/O Kit

Both OS X and iOS only run on specific hardware. This makes that many drivers are shared among different Apple hardware revisions. Drivers are usually written for the I/O Kit framework in the embedded C++ language. Most drivers are closed source, so the attacker is bound to reverse engineering or fuzzing. Besides looking for C++ specific vulnerabilities, I/O Kit also offers multiple ways drivers can communicate with user space applications. Either to set driver specific options, but also to allow helper applications in user space. I/O Kit in the context of security was previously discussed by Van Sprundel[35] and Miller et al.[25]. Because I/O Kit drivers are not as deeply audited as the open source components of XNU there are still a lot of vulnerabilities there.

5.3.3 `ioctl` calls

The `ioctl` call is a special syscall, as its implementation reaches out to many different kernel components. Being an abbreviation of input/output control, `ioctl()` is a single syscall used for setting or retrieving control variables of devices or components in kernel space. For example a call to `ioctl()` could be used to eject a DVD drive or add a rule in the packet filtering chain.

Many `ioctl` calls require moving data from and to kernel space and because so much functionality is added into a single syscall it is error prone, examples are CVE-2009-1235 and CVE-2012-3728. The `ioctl()` system works by calling the `fo_ioctl` function pointer in `fileproc->fg_ops->fo_ioctl`, defined in `bsd/sys/file_internal.h`. Each device has its own method responsible for handling the `ioctl` calls, for example the TTY driver defines `ttioctl()` as his ioctl method in `bsd/kern/tty.c`.

5.3.4 File systems

File systems are complex data structures which usually contain many header fields, header fields with variable sizes and pointers to other data structures. Because file systems are complex but have a fixed data structure, they can automatically be tested by using fuzzing. Creating a valid image of a supported file system and start flipping bits until the kernel crashes is proven to be an effective technique.

A good example of fuzzing is CVE-2012-0642, which is a vulnerability found in the XNU kernel by mounting a corrupted HFS+ image. The vulnerability was found by fuzzing the HFS+ file system by automatically flipping bits in a valid HFS+ image. This eventually led to an image that would consistently crash the kernel.

5.4 Conclusion

There are great differences between targeting XNU on OS X and iOS. Not only the reasons for targeting the XNU kernel differ, but also the attack strategy. Code signing and sandboxing should be taken into account under iOS, where on OS X the attack surface is limited to that of an unprivileged user.

The next chapter will focus on the exploitation techniques an attacker can use after he or she has found a vulnerability. This chapter describes the steps required for bypassing the different mitigation techniques, as well as some specific techniques an attacker can use to gain arbitrary code execution.

6

Exploitation

6.1 Introduction

Chapter 3 and 4 described the different mitigation techniques used by XNU to protect itself as well as the entire operating system against attackers. The XNU kernel features many of those techniques, such as stack canaries and kernel ASLR. But as already stated in those chapters, they can never fully protect the system against an attack. In this chapter we look how an attacker might bypass the mitigation techniques.

In the first section we will show the general steps an attacker might take to bypass the mitigation techniques. In section 6.3 we propose steps an attacker can take to exploit heap vulnerabilities and how an attacker would go from code execution to root privileges.

Based on this chapter we make recommendations on improving the current protection mechanisms. This will be done in Chapter 8, where we discuss the security of XNU.

6.2 Bypassing the mitigation techniques

6.2.1 Kernel ALSR

The need to bypass kernel ALSR can have multiple reasons. Perhaps the attacker needs absolute addresses for his vulnerability, or he needs to find a function or data structure in memory. In order to bypass kernel ASLR the attacker will either need to leak some kernel memory to determine the slide offsets (called an information leak vulnerability), or be able to use a relative offset from his current control point. We describe both methods.

Information leaks

Sources that can leak information about the memory layout are for example log messages and syscalls. The kernel log messages end up in `/var/log/system.log` which is readable for users in the `admin` group. In listing 17 we show that kernel addresses do end up in the log file. If an attacker can determine the base address for one of those addresses, he or she can calculate the slide offset. This attack only works if the attacker is already a member of the `admin` group. Listing 17 also shows that %p (used to print pointer addresses) is still used in the XNU kernel.

```
$ grep kernel /var/log/system.log | \
    grep -E "[[:xdigit:]]{16}" | wc -l
     225
$ grep -R %p xnu-2050.18.24/ | grep -v panic | wc -l
     953
```

Listing 17: leaking kernel addresses

The attacker can also find and use an information leak vulnerability. Three examples are shown in listing 18, 19 and 20.

```
#include <stdio.h>

void secret() {
    int secret = 0xdeadbeef;
}

void leak() {
    int i;

    // will print the secret 0xdeadbeef
    printf("infoleak: 0x%x\n", i);
}

int main() {
    secret();
    leak();

    return 0;
}
```

Listing 18: an example of an uninitialized variable

The `leak()` function of listing 18 forgets to set the `i` integer before it gets printed. Because `i` has no defined value, `i` will hold whatever lies on the stack at that moment. Before the `leak()` function gets called, the `secret()` function gets executed. The `secret()` function will place one secret value onto the stack, which happens to be on the same location as `i` will later be. This is a common type of vulnerability, often seen in syscalls that return a struct to user space where not all struct members get initialized. When looking for an information leak, search for a syscall that returns a struct, which lacks a call to `bzero()` or `memset()`.

The bug in listing 19 is a type confusion bug. The `printf()` method expects an integer, where the second argument is actually a pointer to an integer. This will print the address of where `i` is stored in memory, rather than the value of `i`. The bug in listing 20 let the attacker read arbitrary memory by choosing an index outside the array boundary.

```c
#include <stdio.h>

int main() {
    int i = 0xdeadbeef;
    int* j = &i;

    printf("infoleak: 0x%x\n", j);

    return 0;
}
```

Listing 19: type confusion example

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int array[] = {1, 2, 3, 4, 5};
    int i = atoi(argv[1]);

    printf("info leak: 0x%x\n", array[i]);

    return 0;
}
```

Listing 20: read anywhere example

Information leak vulnerabilities are valuable now that the kernel uses ASLR. Before ASLR leaking kernel addresses was not considered harmful and no one payed attention to this type of vulnerability. Many syscalls even returned kernel addresses as part of their specification. For example the `kextstat` utility printed the base address for all kernel extensions. With the release of kernel ASLR Apple tried to close all known address leaks. In the case of `kextstat` this is done by unsliding the address before returning, so no useful information leaks to the user. However we expect to see a lot more patches for information leaks in the future. The first information leak vulnerability is already a fact (CVE-2012-3749).

Using a relative offset

The attacker might be able to overwrite anything relative from his location to gain control. An example is shown in listing 21. The attacker can overflow the function pointer relative from its location, giving him control over the instruction pointer. He or she can use this to point the instruction pointer to a mapped memory page in user space for example.

Once the attacker has gained arbitrary code execution he or she can use the values stored in the CPU registers to leak the slide offset, for example by searching for a return address on the kernel stack (which should be located on `rbp + 0x8`). This address can be compared to the address in the kernel binary to calculate

```
$ cat example.c
#include <stdio.h>

void test() {
    printf("hello!\n");
}

int main(int argc, char** argv) {
    void (*funcptr)() = &test;
    int list[1];

    list[atoi(argv[1])] = atoi(argv[2]);

    funcptr();

    return 0;
}

$ ./example 0 0
hello!
$ gdb -q ./example
Reading symbols for shared libraries .. done
(gdb) r 1 1094795585
Starting program: /Users/user/example 1 1094795585
...
Reason: KERN_INVALID_ADDRESS at address: 0x0000000141414141
0x0000000141414141 in ?? ()
(gdb)
```

Listing 21: a relative overwrite

the slide offset. In section 6.3.2 we propose a method to find the `proc` struct in memory using a relative offset.

6.2.2 Stack canaries

Stack canaries offer protection against basic stack smashing, where the attacker would overwrite the stored instruction or frame pointer. The attacker cannot brute force the cookie (which is a technique used in threaded user space applications), since the kernel will panic immediately. Turning a stack buffer overflow into successful code execution is now limited to two options, described below.

The first method for exploiting a stack based buffer overflow is overwriting any local variable on the stack, instead of the stored instruction pointer. As can be seen in figure 6.1, overflowing the buffer would first overwrite a function pointer before reaching the canary value. This attack depends on the local variables that are located below the vulnerable buffer on the stack. There might be a function pointer, or any other variable that, when altered, gives the attacker new possibilities.

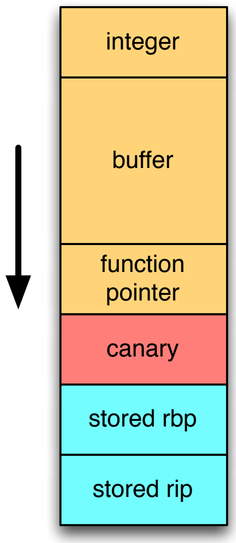


Figure 6.1: overflowing buffer will give control over the function pointer

The second method is to first find an arbitrary read vulnerability, which allows the attacker to first read the stack cookie so that the attacker can overwrite the cookie with the right value. Since the second byte of the stack cookie is always 0, this technique will fail if the vulnerability is caused by a string function.

6.2.3 WˆX protection

We work on the assumption that the attacker is able to control the instruction pointer in kernel space, by exploiting any vulnerability.

WˆX protection in user space is typically bypassed by using Return-Oriented Programming (ROP). Because building a ROP stack is a difficult and time consuming task, the exploit is often divided into multiple stages. The first ROP stage is used to map a new page into kernel memory with write and execute permissions, after which shellcode from a known location is copied into the new page. Then execution is continued from the newly created page. This technique requires only a few function calls using ROP.

Another option is to reroute execution to an already write and executable page in memory. On OS X the attacker can map its own page with shellcode in user space memory and reroute execution to user space. For iOS this technique is probably not possible, since the attacker cannot map a new page due to the code signing enforcement. This technique would only work on iOS if the attacker has control over the JIT page of the Mobile Safari browser.

6.2.4 Code signing

Kernel exploits are typically complex and large in size. On iOS, where code signing is always enabled, this can become a problem when exploiting a kernel vulnerability. Strict code signing restrictions leave the attacker with no other option than using ROP for his attack. Staging an entire kernel exploit using only ROP gadgets might not be possible. Exploits that require branching and looping are difficult to implement using only ROP gadgets.

If an attacker targets the iOS version of XNU, having a vulnerability that is easy to trigger and can have a payload in an external data format can have considerable advantages. An example is the jailbreak for iOS 5.0.1 which used a vulnerability in the HFS+ file system format. Triggering this vulnerability was done by a sequence of `mount()` and `umount()` calls using ROP. The mounted images contained the payload for the exploit.

On OS X with Gatekeeper enabled the attacker could map a memory page with `rwx` permissions inside another application by exploiting it. From here a normal kernel exploit can be launched.

6.2.5 Sandboxing

Sandboxing can limit the syscalls an application is allowed to call. In such a situation the attacker has two options, either break out of the sandbox or find a vulnerability he or she can exploit from within the sandbox.

Breaking out of the sandbox can be done by taking over a process with more privileges than the current process. This can for instance be done by dropping a file that another process will read later or sending an IPC message to a vulnerable service. On OS X and iOS all applications have access to the Mach Bootstrap port, which is used as a lookup service for other Mach RPC services. There are more than 140 RPC servers providing various services such as a pastebord. All services are accessible for every applications. The attacker could exploit one of those services that has a less strict sandbox.

Some vulnerabilities can be exploited without the need to break out of the sandbox. This can be because the vulnerability is in a syscall that is allowed in the current sandbox profile, or access to the syscall cannot be restricted by the sandbox. Each syscall has to explicitly call the MAC framework by calling the `MAC_CHECK()` macro, which some syscalls neglect. An example is the `ptrace()` syscall. A sandboxed process that runs as root, was allowed to `ptrace` another non sandboxed root process.

The sandbox implementation is effective in reducing the attack surface. However its effectiveness depends entirely on the sandbox profile thats being used, which has to be as strict as possible. Our Pwn2Own submission illustrates this: a single WebKit vulnerability was sufficient to extract near all personal information from the device (photo, video's, browsing history and the address book). To be effective, the sandbox profile needs to prohibit all access to disk, network and syscalls that are not required by the application.

6.3 Exploitation

6.3.1 Attacking the zone allocator

Debugging the zone allocator

Sometimes the origin of a heap corruption is uknown, for example if the attacker used fuzzing to find the vulnerability. The first step in exploitation is to find the origin of the corrupted allocation. The XNU kernel has two mechanisms to debug the zone allocator, which can lead to the origin of a corruption. The mechanisms can be enabled by setting a boot argument: `sudo nvram boot-args=""`.

Adding the boot parameter `-zp` will poison (fill) each zone element with the magic marker `0xdeadbeef` when it is freed. When printing the free elements, the corrupted zone element can be detected by having an corrupt magic marker. This method will not show the allocation who was responsible for the corruption, but does show in which block the corruption takes place. It can be that the previous block has overflown, or that there was still a reference to the freed block (a use-after-free situation).

The zone allocator also support zone corruption logging, which is enabled by the boot parameter `-zc`. The `-zc` parameter should be combined with the `zlog=<zone_name>` parameter. Zone corruption logging will keep a back trace of all allocation and deallocations in the specified zone. The macro script for `gdb`, which is included in the kernel debugging kit, can be used for printing the logged information. Such as `zstack` for showing the zone call stack and `showzalloc` for a list of allocations in the `zalloc` table. Use `help kgm` for a list of all available commands.

Heap Feng Shui

In a typical heap buffer overflow the attacker can overflow a buffer stored on the heap into an adjacent heap block. To do anything more than just crashing the kernel, the attack needs to be able to control which data will lie in the adjacent block. Normally the adjacent block is difficult to predict, since the heap gets fragmented after a while. Therefore the attacker needs a method which will increase hiss odds on successfully predicting the next allocation. A common used technique for controlling the contents of the adjacent block is Heap Feng Shui presented by Sotirov[33]. While his work covered the heap allocator used by Internet Explorer, we show that the techniques can also be applied to the zone allocator, due to its deterministic allocation algorithm.

Figure 6.2 shows the involved steps for a successful Heap Feng Shui attack in the zone allocator, with the description below:

1. The initial state. Over time the zone has been fragmented. The attacker has no information on where the free blocks lie in the zone, nor which block is on the top of the free list.
2. The attacker first has to fill the zone with allocations. It is important that these allocations will end up in the same zone as the zone of the vulnerable

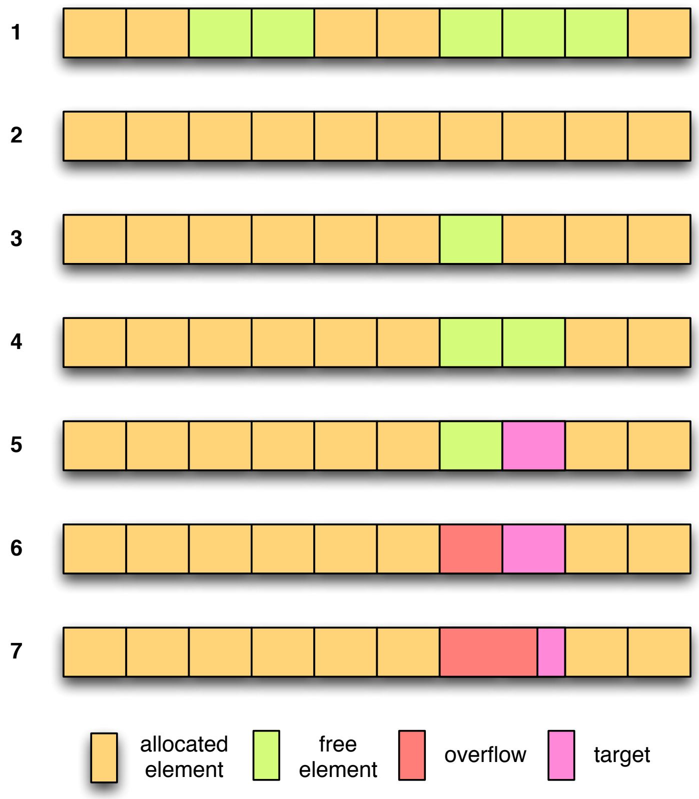


Figure 6.2: Heap Feng Shui illustrated

allocation. Since zones are shared between kernel threads, using a quiet zone is recommended if the vulnerability allows this. Picking a quit zone will reduce the risk of interference of other threads.

For example an exploit for CVE-2012-0642 (a HFS+ vulnerability) used a HFS+ image with a block size of 512 bytes, instead of the default of 4096 bytes. The 512 bytes zone is much quieter than the 4096 bytes zone since this one is also used by other file systems on the system.

3. After the zone is filled with allocations under control of the attacker, the next step is to punch a hole in the zone by freeing one of the later allocations. The later allocations will be more likely to be adjacent to each other, as fragmentation is more likely to occur at the beginning of the zone.
4. A second block is freed, which was allocated right after the block of the previous step. This will give two free blocks that are adjacent to each other. This block will form the top of the free list.
5. The target block is allocated. This block will hold something valuable to overflow in, such as a function pointer.

6. The following allocation will be right before the target block. This allocation will hold the element that will overflow.
7. The final step. The attacker triggers his vulnerability, overflowing in the adjacent block holding the target element.

(Ab)using the next pointer

Besides overflowing in an adjacent allocated block, the attacker could also overwrite the next pointer of an free block. This will give the attacker control over a pointer that is returned by `zalloc()`. The `zalloc()` function will pick the current head of the free list, which is the address of an free block in the zone. From this free block `zalloc()` will read the first 4 or 8 bytes (depending on the architecture), which holds a pointer to the next free block. The old head of the free list is returned, and the pointer that was just retrieved from the free block is the new head of the free list. If the attacker can control a next pointer, this pointer will be returned by `zalloc()` at some point. The calling function will start writing data to this location. The attacker should make sure that he controls the data that is written.

If the attacker is able to both overwrite the next pointer and controls the data that is written to the corrupted block, he can write to arbitrary memory location. Because the next pointer is an absolute address, precise knowledge of the memory layout is needed, meaning the attacker needs to bypass ASLR before this attack.

6.3.2 The post-exploitation step

An attacker that targets the kernel has a certain goal in mind. This goal might vary depending on the platform. For OS X this might be gaining root privileges, where for iOS it might be to disable the security restrictions. After the attacker has a stable exploit, he needs to patch relevant kernel memory to achieve his goal. We call this the post-exploitation step. In the following section we propose a way the attacker can elevate his privileges and briefly describe the technique used by jailbreakers.

OS X

On OS X an attacker would target the kernel to gain root privileges. If the attacker manages to successfully stage his attack to the point he has full control over the program flow, the next step would be to locate the `ucred` struct for the current process.

The `ucred` struct holds the user ID, changing the user ID for the current process to 0 will give it root privileges. The `ucred` struct is embedded in the `proc` struct of the current process. The attacker will first need to find his current `proc` struct in memory.

When a new process is created, for example by the fork() or vfork() syscall, the cloneproc() routine is called (defined in bsd/kern/kern_fork.c) for the actual process creation. The cloneproc() routine will first call forkproc(), who is in charge of creating the new proc struct (but not create the actual process). The proc struct gets stored in a zone which is created by MALLOC_ZONE(). After forkproc() is done, cloneproc() will call fork_create_child(), who will create a Mach task and thread for the new process. The newly created task and thread struct will be stored in the task_zone and thread_zone respectively.

All structs that are directly relevant to the attacker are placed in a specific zone of the zone allocator. The memory offsets of the allocations are unknown to the attacker due to both ASLR and the fragmentation of the zone.

To find its proc struct the attacker could use a BSD sycall, as each BSD syscall receives a pointer to the proc struct of the calling process as its first argument. This means that if a BSD syscall is used, a pointer to the current proc struct will be somewhere on the kernel's process stack. The offset from a CPU register pointing to the stack, and the location of the pointer to the proc struct will be a fixed value and can thus be used. An example is shown in listing 22.

```
; obtain the proc struct (first argument)
mov    rcx, [rbp + 0x10]
; obtain the ucred struct
mov    rax, [rcx + 0x64]
; set cr_uid
mov    [rax + 0x8], 0
```

Listing 22: setting uid to 0

iOS

On iOS the attacker wants to disable the code signing enforcements in the kernel. This can be done by patching pieces of the source code in memory to remove the checks that are done regarding the signatures. Before the attacker can overwrite the code section he first needs to add write permissions to the memory page by for example calling vm_protect(). To be able to patch kernel memory the attacker needs to be able to write to arbitrary places in memory.

In the previous jailbreaks the attackers would use an write primitive to patch the sysent table to add an additional syscall. Calling the syscall from user space would trigger the payload, which patched the necessary kernel routines. The patches that are needed for a jailbreak are described by Miller at al.[25].

6.4 Conclusion

In this chapter we showed how an attacker might bypass the active mitigation techniques. Based on this knowledge we form our recommendations on further hardening the kernel. As we have shown, depending on the situation, the mitigation techniques can be bypassed by an attacker. Mitigation techniques that can fully eliminate a certain vulnerability class is rare. An example in XNU is `NULL` pointer dereference protection, which completely mitigates the risk of exploitation for this vulnerability class.

We proposed concrete ways to determine the slide offset, attacks against the zone allocator and ways to locate the `proc` struct in memory.

This was the final chapter about the security of XNU. We discussed the architecture of XNU and its mitigation techniques. In this chapter we described the steps an attacker might take to attack the kernel. The next chapter we will compare the XNU kernel with other kernels and the mitigation techniques they implement.

Part IV

Evaluation and comparison

7 Comparison

7.1 Introduction

In the previous chapters we looked at the implementation of security features in the XNU kernel. We stated that the XNU kernel has effective hardening features such as kernel ASLR and WˆX protection, which require a higher skilled attacker to bypass. In this chapter we compare XNU with other kernels in the context of security, to compare the effort Apple has invested into kernel hardening but also to see if there are other hardening techniques from which XNU could benefit.

For the comparison three kernels were chosen: the FreeBSD, Windows an Linux kernel. The FreeBSD kernel because a large portion of the XNU kernel was originally build from FreeBSD kernel code. Hardening techniques in todays FreeBSD version might be easy to adapt and implement in the XNU kernel or the other way around. XNU has benefit from FreeBSD hardening projects in the past, for example the TrustedBSD project, which is used for code signing and sandboxing in XNU, was originally started as a FreeBSD hardening project.

The second kernel we compare XNU to is the Windows kernel. Windows is Apple's largest competitor on the desktop market where it also has the highest market share. Windows and OS X have a completely different architecture as Windows does not follow the UNIX philosophy. However, most exploitation techniques are used across different platforms (such as ROP for example). Microsoft was forced to improve the overall security of their operating system as they are frequently targeted by malware, this has resulted in a secure operating system. XNU could benefit from the hardening techniques developed by Microsoft.

Lastly we compare XNU with the Linux kernel. The Linux kernel is, apart from the different desktop distributions, used to power the Android operating system, the biggest player in the mobile device market. Linux is based on the UNIX operating system, which makes that it has a lot in common with the FreeBSD and XNU kernel, although they don't share a code base.

For the comparison we look at the implemented protection mechanisms for each kernel. Do they use the same techniques as XNU, such as stack canaries and kernel ASLR, or do they also use additional techniques and could they be relevant for XNU.

7.2 FreeBSD

The development of the FreeBSD operating system started in 1993 as a fork of 386BSD. The operating system rapidly gained success, especially under hi-traffic websites such as Yahoo! and Hotmail. Version 2.0 featured the virtual memory system from Mach, which is also used in XNU. The latest release at the time of writing is version 9.0 with version 9.1 awaiting its release.

7.2.1 Memory protection

System hardening is no priority for FreeBSD. Not only does FreeBSD has limited kernel protection mechanisms, but also user space is missing many protection mechanisms that are common on other operating systems for years. For example, user space applications have an executable stack, because it must hold trampoline code for calling and returning from a signal handler (this is fixed in the yet to be released version 9.1). Currently there is no ASLR implementation and all applications outside the base system are compiled without GCC's stack protector. The lack of hardening techniques led to the implementation of custom patches by FreeBSD users, but they are yet to be merged with the main source tree.

User space is missing most of the memory protection mechanisms that we find common on other operating systems, and the same is true for the kernel. Until the release of FreeBSD 8 (in 2009), the kernel had no hardening techniques that should protect memory from corruption. With the release of FreeBSD 8 stack and `NULL` pointer dereference protection was added.

The FreeBSD kernel is compiled using the GCC compiler (although effort is being made to switch to the Clang compiler instead), which comes with ProPolice for stack smashing detection. ProPolice will protect all stored CPU registers (such as the stored instruction and base pointer) by placing a canary value between local variables and stored CPU registers on the stack. The canary value is generated by using `arc4rand()` (libkern/arc4random.c), which is called by `__stack_chk_init()` in `kern/stack_protector.c`. It will generate an 8 byte random number by using the tick counter as seeding value. The canary value does not necessarily contain a 0 byte (except when the random function will generate a 0 byte). ProPolice will also try to reshuffle variables on the stack, such that arrays will be located on higher addresses than local variables of another type. This should prevent the possibility of overwriting important local variables with an stack based overflow.

The risk of `NULL` pointer dereferences is mitigated by disallowing the mapping of the first page of memory. It is enabled by default, but can be disabled at will by setting the `security.bsd.map_at_zero` `sysctl` value.

7.2.2 Advanced mitigation techniques

For hardening the overall system security, a system administrator can enable FreeBSD's secure level. Secure level does not protect against memory corruption but restricts some of the core components when set. The secure level is disabled by default, but can be assigned a number between `-1` (disabled) and 3 (the most strict). The restrictions that apply are (taken from the `security(7)` man page):

- -1. Permanently insecure mode - always run the system in insecure mode. This is the default initial value.
- 0. Insecure mode - immutable and append-only flags may be turned off. All devices may be read or written subject to their permissions.
- 1. Secure mode - the system immutable and system append-only flags may not be turned off; disks for mounted file systems, `/dev/mem` and `/dev/kmem` may not be opened for writing; /dev/io (if your platform has it) may not be opened at all; kernel modules (see `kld(4)`) may not be loaded or unloaded.
- 2. Highly secure mode - same as secure mode, plus disks may not be opened for writing (except by `mount(2)`) whether mounted or not. This level precludes tampering with file systems by unmounting them, but also inhibits running `newfs(8)` while the system is multi-user.

  In addition, kernel time changes are restricted to less than or equal to one second. Attempts to change the time by more than this will log the message "Time adjustment clamped to +1 second".
- 3. Network secure mode - same as highly secure mode, plus IP packet filter rules (see `ipfw(8)`, `ipfirewall(4)` and `pfctl(8)`) cannot be changed and `dummynet(4)` or `pf(4)` configuration cannot be adjusted.

Enabling secure level goes by setting the `sysctl kern.securelevel` value. The security level cannot be altered by the root user without rebooting in single user mode, and since loading kernel extensions is disabled after setting, a kernel exploit is needed when an attacker needs to lower the security level on a remote system.

An external project which tries to improve FreeBSD's security is the TrustedBSD project[1]. the TrustedBSD MAC framework is used for code signing and sandboxing in XNU, but was originally developed for FreeBSD. TrustedBSD is an ongoing effort to bring advanced security modules to FreeBSD. Not all TrustedBSD patches are adopted by the FreeBSD kernel, but most patches will find their way to the main source tree. Examples are the SYN cookies, GEOM and the security audit system.

With the release of FreeBSD 9.0 (in 2012) experimental support for the Capsicum framework[2] was added. The Capsicum framework is a sandbox framework developed by the University of Cambridge and supported by Google. The Capsicum framework is capable to limit access rights of a process, such as syscalls, similar to that of XNU.

FreeBSD has, with secure level, TrustedBSD and the Capsicum framework some interesting hardening techniques. Though they are not meant to protect the kernel, but rather the system as a whole. If we focus on memory protection

[1]trustedbsd.org
[2]cl.cam.ac.uk/research/security/capsicum

we have to conclude that FreeBSD is missing many of the techniques that are standard today.

7.2.3 Security research

FreeBSD is a small player compared to the other operating systems. It is mostly used to power servers, where it seems to loose market share to Linux based system. It is not used in any popular mobile operating system. Its small user base has its effect on the level of security research. The FreeBSD kernel has a low number of vulnerability reports, with only a couple of advisories each year. Documentation about the FreeBSD in the context of security is outdated.

However there are some community projects that try to improve the level of security the FreeBSD kernel has to offer. The TrustedBSD and Capsicum frameworks are examples of this. By sandboxing all high risk applications (applications that listen on an external interface for example) the system is well protected against most attackers, despite the lack of memory protection mechanisms.

7.3 Linux

Linux is an open source clone of the UNIX kernel. Strictly speaking Linux is only the kernel, though the term Linux is often used to refer to a complete operating system. The Linux kernel is ported to many different platforms, with currently supporting eighteen different architectures. At the time of writing the current stable release is 3.6, though most distributions will run on an older release.

7.3.1 Memory protection

The Linux kernel shares its virtual address space with a user process, just as the newer OS X releases. Sharing the address space with user space processes enables a commonly used exploitation technique, where the attacker would use a kernel vulnerability to let the the program counter point to user space memory, continuing execution with kernel privileges from attacker controlled memory. Intel has implemented a CPU feature called Supervisor Mode Execution Protection (SMEP)[3], which eliminates this technique by preventing the execution of kernel code from use space memory. The Linux kernel supports this new feature, which means the attacker has to find a way to copy its shellcode into kernel space in an executable page, or use ROP instead[30].

The kernel map is protected by WˆX, which protects the execution from code on the stack and heap, as well as the alteration of important structs and kernel code. The NX feature is enabled if it is supported by the CPU, otherwise a software NX implementation is used. To defeat a system with both SMEP and WˆX enabled the attacker could use a newly described technique which uses

[3]intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf

the JIT compiler feature of the Berkeley Packet Filter (BFP), as described by K. McAllister[24]. Filters of the BFP run inside a virtual machine in kernel space, for performance reasons. User space applications can supply their own filter which will get executed, making that the attacker can use traditional JIT spraying techniques[8] to gain code execution without the need for ROP.

Stored CPU registers on the stack are protected by GCC's stack protector, which also shuffles variables to protect local variables. The stack canary is generated during boot which is architecture specific. For ARM processors (the main architecture for Android), the canary is generated in `arch/arm/include/asm/stackprotector.h`. It will generate a 4 byte canary, with no 0 bytes. The random entropy is proved by various sources (such as network packets).

Kernel heap allocations are handled by the SLUB allocator, which is a descendant of the SLAB allocator. Currently the SLUB allocator offers no decent protection against attackers. The SLUB allocator had an inbound free list, like XNU's zone allocator which can be tampered with by an attacker. The free list is not protected by for example a canary value. Furthermore there is no protection for overflowing in adjacent allocated blocks.

`NULL` pointer dereference vulnerabilities are mitigated by disallowing allocations of the first 64 KB of memory. This value is controlled by the `vm.mmap_min_addr sysctl`, and can be changed by an administrative user to a higher value (or lower, but this is highly discouraged).

7.3.2 Advanced mitigation techniques

The Linux kernel has an extension mechanisms called Linux Security Modules (LSM)[40], which allows a module to hook into important kernel event where user data might influence important internal kernel structures. Linux Security Modules act as an access control mechanisms in these situations. Examples of LSM modules are SELinux and AppArmor, which implement a sandbox model for applications.

Some security frameworks decide to patch the kernel instead of using the LSM framework. This might be necessary when the LSM framework does not support all the required features. The grsecurity project[4] is an example of this. The grsecurity project tries to harden the overall security by applying patches to the Linux kernel. For example, grsecurity restricts access to the `proc` file system, which is otherwise used by attackers to gain information about the memory layout for example. But also various kernel hardening techniques are implemented, such as kernel stack ASLR.

Which protection mechanisms are enabled, depends greatly on the chosen Linux distribution. Some distribution ship an older version of the kernel, which has less available security features, where other distributions might choose to disable them for various reasons. The Ubuntu Linux distribution has a special website dedicated to kernel security and all hardening techniques they support[5].

[4]grsecurity.net
[5]wiki.ubuntu.com/Security/Features

7.3.3 Research

The Linux kernel is a true open source development project, which gives users full read access to the latest development tree. There is an active community investigating the security of the Linux kernel. This research has not only resolved vulnerabilities, but has also brought some new mitigation techniques, such as the grsecurity project from the previous section.

Others have presented their work on bypassing the mitigation techniques in the Linux kernel. So is the article by K. McAllister[24] an investigation on bypassing SMEP and W^X. D. Rosenberg gave numerous presentations about the security of the Linux kernel, for example on the topic of exploiting the SLOB allocator[31] and how to bypass the restrictions posed by grsecurity[32].

7.4 Windows

The first version of Windows was released in 1985, as an add-on for Microsoft's MS-DOS operating system. It soon became the largest operating system on the desktop market. At the time of writing Windows 8 has just been released. Due to the high market share Windows is a beloved target of malware writers. This has led to the implementation of many mitigation techniques in the Windows operating system, both in user space as well as in kernel space.

7.4.1 Memory protection

Visual Studio comes with a stack protection features called `/GS`, which is not only enabled for user space but also for the kernel. Windows 8 is build using Visual Studio 2010, which brings several enhancements to the old `/GS` implementation. A random canary is generated and for the 64-bits kernel the most significant two bytes will be set to `NULL` to stop the various string functions. Local variables are reordered and the compiler inserts automatic array bound checks to completely mitigate certain vulnerabilities. The compiler will try to detect the use of insecure copy functions, and replace them with copy functions that do bounds checking.

The kernel pool is the heap allocator used in the Windows kernel. It stores meta data inbound, and thus vulnerable for tampering. The kernel pool implementation is discussed in depth by others[22][28][5]. With the release of Windows 7 and 8 Microsoft tries to protect the integrity of the inbound stored meta data. Pointers are now protected by canaries, similar to stack canaries. Furthermore pointers are verified to be located inside the heaps memory map and some inbound stored meta information is moved to a look-a-side buffer. This has led to a much more secure heap implementation in Windows 8. The design of the kernel pool differs greatly from that of the zone allocator, making it difficult to compare the two. The kernel pool makes extensive use of canaries to protect against attackers. Though the kernel pool is still vulnerable to some attacks as shown by Mandt[1].

All memory pages of the Windows kernel are protected by WˆX. Windows refuses to install on machine's without support for hardware NX (which is a CPU feature). Just as in XNU, the Windows kernel shares its virtual memory space with user space processes. To make jumping to user space from kernel context (to circumvent WˆX) impossible, support for Intel's SMEP architecture was added.

Just as XNU, the Windows kernel uses ASLR. Under Windows 7 the entropy for ASLR was 5 bits. Under Windows 8 this has been increased to 22 bits of entropy obtained by the `rdrand` instruction when possible. Windows 8 also fixes some known information leaks in various syscalls, so attackers have to find new ways of leaking kernel memory to defeat ASLR.

`NULL` pointer dereference vulnerabilities are now also mitigated by prohibiting the mapping of the first memory page. This is similar to the mitigation found in other operating systems.

7.4.2 Advanced mitigation techniques

When looking at the more advanced mitigation techniques we see support for several techniques that try to keep the integrity of the kernel, with support for UEFI's secure boot option, driver code signing and patch protection.

The secure boot option is similar to that of iOS. The UEFI is shipped with a list of accepted signing keys and checks the boot loaders signature before handing over the boot process. This should limit the risk of bootkits infecting machines. Secure boot is a new technique which will have to prove itself. Unlike iOS a PC can boot multiple operating systems, as a result there can be multiple signing keys. To prevent attackers to use stolen signing keys, keys can be placed on a revocation list. This revocation list however needs to be updated by the UEFI on a regular basis, otherwise stolen keys can be used to boot an infected kernel.

After the kernel is loaded it only accepts signed kernel extension. For signing an extension a developer needs to request a code signing certificate from an approved certificate authority. Developers with a valid certificate can sign their own extensions, without approval from Microsoft. This enforcement has led to attackers stealing valid signing certificates from hardware manufactures. The Stuxnet virus for example used two stolen signing keys from RealTek and JMicron to infect systems[12]. The code signing enforcement can be disabled by an administrative user for testing purposes (which is not recommended for production machines).

Kernel Patch Protection (KPP) should reduce the risk of rootkits compromising a machine. It was first introduced in the x64 version of Windows XP. KPP works by periodically check protected structs for tampering. Upon detection the kernel will panic. Attackers who use a kernel exploit to gain administrative access might have no problems with KPP, as it is meant to prevent rootkits. If an attacker restores all his alteration to kernel memory after a successful exploit, it might go unnoticed by KPP. To prevent attackers from simply disabling KPP it is highly obfuscated and nested deeply into the roots of the kernel. An in depth description of KPP was given by Gupta et al.[3].

7.4.3 Secure development practices

Microsoft has clear guidelines on secure development, which they call the Security Development Lifecycle (SDL). During the development of Windows 8 Microsoft used SDL on all development projects. SDL is a seven phase development process which should reduce the number of vulnerabilities by making security an integrated part in each step of the development process. The seven phases in SDL are:

1. **Training**
   Train developers about secure programming.
2. **Requirements**
   Define the minimum security and privacy criteria, this includes a risk assessment.
3. **Design**
   Analyze the attack surface and threat model of the application. Design the application with security in mind.
4. **Implementation**
   Use static analysis to detect vulnerabilities before compilation. Only use approved tools and deprecate unsafe functions.
5. **Verification**
   Review the attack surface and perform automatic testing by using fuzzing and dynamic analysis.
6. **Release**
   Create an incident report plan and perform a final security review.
7. **Response**
   Respond to reported vulnerabilities.

The Windows kernel was designed before the introduction of SDL, but some steps can be done retroactively on older kernel code, such as static and dynamic analysis. For new or refactored components of the kernel the SDL practices apply. More information about SDL can be found on Microsoft's website[6]

7.4.4 Research

Windows is entirely closed source, so enhancing the security by patches (such as TrustedBSD on FreeBSD and grsecurity on Linux) is difficult. Furthermore the signing requirements of kernel extensions and the implemented patch protection limits the possibility for security enthusiasts of implementing additional hardening techniques. Hardening techniques are likely to be implemented in user space under Windows, rather than in kernel space.

Examples of additional hardening techniques are the sandbox implementations of Google Chrome[7] and Sandboxie[8]. They both don't rely on an additional

[6]microsoft.com/security/sdl
[7]dev.chromium.org/developers/design-documents/sandbox
[8]sandboxie.com

kernel extension, but run entirely in user space.

The security of Windows is well researched. Numerous presentations cover the topic of Windows kernel security[22][1]. Microsoft also informs the security industry with information about new hardening techniques implemented in the latest version of Windows[17].

7.5 Conclusion

Currently Windows has the most implemented mitigation techniques. But it is not only the number of mitigation techniques that make that Windows stands out from the rest. Microsoft has clear guidelines for secure developments which include static and dynamic analysis of the source and making security an integral part of the development process. Adding security at the end of a development project is costly as it often requires refactoring of the code base. That security has to be a part of the design process is something the security industry has known for years, but is still being ignored in most development projects. Microsoft takes a stand by enforcing in house development projects to adopt their SDL practices.

	XNU	**Linux**	**Windows**	**FreeBSD**
WˆX	X	X	X	-
Stack canaries	X	X	X	X
Heap protection	X	X[1]	X	-
Kernel ASLR	X	-	X	-
`NULL` pointer protection	X	X	X	X
Patch protection	-	-	X	-
SMEP	-	X	X	-
Code signing	X	-	X[2]	-
Sandboxing	X	X[3]	-	X[4]

[1] though it is more meant for detecting bugs than actual hardening
[2] only for kernel extensions
[3] by using AppArmor or SELinux
[4] by using Capsicum or TrustedBSD

Table 7.1: comparison of protection mechanisms

A comparison of all mitigation techniques for each operating system is shown in table 7.1. Windows has implemented most of the hardening features, whereas FreeBSD has almost no mitigation techniques in its kernel (nor in user space for that matter).

Comparing XNU to other kernels shows that XNU has most of the mitigation techniques in place. It lacks SMEP support for OS X, which is a new technique. It might be possible that Apple will add this feature in the future. Active heap protection as Windows might be possible, but has a performance impact that might prevent Apple from enabling it right now, where Microsoft chose security over performance. As for patch protection, this might be a appropriate technique for preventing the jailbreaking of iOS, as jailbreakers patch various kernel routines to disable code signing. Patch protection will have its impact on performance however, as kernel memory has to be inspected for evidence of tampering on a regular basis.

Following a clear set of guidelines for secure development may be the most beneficial for XNU. Microsoft SDL practices include fuzz testing for example, which can prevent a lot of security vulnerabilities to be found by outsiders. The kernel often operates on large data sets, such as file systems, which would be ideal for fuzz testing.

8 Conclusion and discussion

8.1 Introduction

We presented the current security state of XNU, which protection mechanisms are in place and how they are implemented. We see that Apple started with adding hardening features to XNU in the later version of their operating systems. Looking at the industry we see a trend in the interest in iOS security. Since 2011 numerous talks about iOS security were given at the various security conferences, not only covering XNU but the entire iOS operating system. The same trend we see in public XNU exploits, which for the past few years all require root privileges for triggering, meaning they are only of interest for attackers targeting iOS. It might be that jailbreaking is the reason Apple started to harden the XNU kernel.

For OS X we see no such trend, nor in OS X security in general. Except for the Flashback malware outbreak in April 2012. Flashback infected over 550,000 machines[39], by exploiting a vulnerability in the Java Applet plugin. The vulnerability was patched by Oracle in February 2012. Apple, until then, distributed its own version of Java, which was not patched until after the outbreak. Flashback did not use any privileged escalation vulnerability, and thus other users on the same system were not infected. We are unaware of any recent research on local exploitation of OS X.

In section 1.4.2 we presented our four research questions. We start this chapter with answering those questions. Or opinion on the security of XNU is left for the discussion in section 8.3.

8.2 Research questions

1. What security mechanisms are currently implemented and how effective are they against exploitation?

Both the iOS and OS X version of XNU come with kernel ASLR, W^X protection, stack cookies, `NULL` pointer dereference protection and limited heap protection. Those techniques try to prevent successful exploitation by either detect memory corruption before the attacker could gain control (stack cookies and heap protection are an example of this), or by making the process from control to code execution difficult (examples are kernel ASLR and W^X protection). These techniques do not try to close existing vulnerabilities, but rather to make exploiting them harder or impossible.

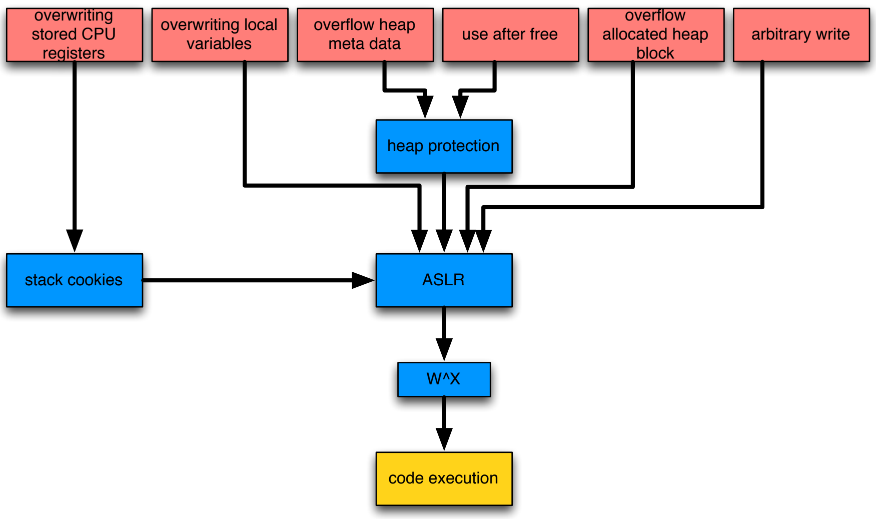


Figure 8.1: protection mechanisms and the bugs they try to prevent

Figure 8.1 gives an overview of the most common type of attacks and the protection mechanisms that have to be bypassed in order to gain arbitrary code execution. As can be seen, half of the attacks are only mitigated by ASLR and WˆX protection. The memory protection mechanisms in todays version of XNU will not stop a motivated attacker, though in some situations multiple vulnerabilities might have to be chained together to bypass all mitigation techniques.

The stored CPU registers on the stack are protected by a random canary value. Stack canaries can prevent a stack based vulnerability from being exploitable and thus is an effective mechanisms. This has manifested in more heap based exploits if we look at the recently used kernel exploits in the jailbreak scene. If the attacker is unable to overwrite a local variable that gives control, the stack canary protection can only be bypassed if the attacker is able to read and write the canary value. This requires both a read vulnerability and a stack vulnerability which does not include a string function (since the canary value will always contain a 0 byte).

The zone allocator uses an inbound free list. To protect the free block from tampering a poison value is added to a free block, and occasionally a copy of the next pointer is added to the end of a block. The effectiveness of heap protection depends on the freedom a vulnerability gives to the attacker. A free block is now filled with a magic marker, and an optional safe copy of the next pointer at the end of a block. The attacker must be able to write those values. Though the attacker will in most situation be able to exploit the vulnerability in some other way, for example by overwriting an allocated block instead of altering the next pointer of a non allocated block.

The ASLR implementation is new and only effective if the attacker can not obtain any information about the current memory layout of the kernel. Because this is a new feature and the leaking of kernel addresses was no issue before this feature (some syscalls even returned kernel addresses as part of their specifications), the kernel will still leak addresses on many places. Apple already tried to patch all known leaks by obfuscating pointers returned by the kernel, but the first CVE (CVE-2012-3749) for an information leak in XNU is already a fact.

XNU sets strict permissions on memory pages to prohibit tampering and execution from pages that control user data, called WˆX protection. To bypass WˆX the attacker can fall back to a traditional ROP exploit in kernel space or reroute the execution to a user space memory page which is under control of the attacker. We see no situation in which WˆX would prevent a vulnerability from being exploitable. It does require a more sophisticated exploit from the attacker.

IOS has a strict code signing enforcement. On OS X code signing can also be enabled, but is less strict than the iOS version. By enforcing code signing the attacker cannot introduce new code, such as a kernel exploit. As a result a kernel exploit needs to be staged using existing code from a signed binary (ROP). Converting a kernel exploit to a ROP version takes time and the resulting exploit is specific for a binary. Code signing is an effective measure for protecting both the integrity of the system as well as protecting the kernel.

Sandboxing user space applications reduces the risk of an attacker compromising an entire system remotely. Because the attacker is unable to read and write arbitrary files, or to call arbitrary syscalls the attack surface of the system is smaller. To be of the most effect all applications on a system should be sandboxed as strict as possible. Sandboxing is an effective technique for both hardening the kernel and the rest of the system.

A sandbox in only effective if the attacker is unable to obtain any useful information from within the sandbox. For our Pwn2Own submission we exploited the Mobile Safari browser and extracted all photo's from the device. This could be easily prevented by using a more strict sandbox profile.

2. How much protection does the XNU kernel offer when comparing it to other kernels?

In Chapter 7 we made a comparison between the different hardening techniques used by different kernels. The comparison showed that the XNU kernel has invested more time in system hardening that its ancestor, the FreeBSD kernel. It has more mitigation techniques than the Linux kernel and defeats the Windows kernel with its build-in support for code signing and sandboxing. Though Windows has more effective heap hardening, patch protection and follows a clear set of secure development guidelines.

Looking at the memory protection techniques we conclude that XNU would benefit from the heap hardening techniques that have been implemented in the Windows kernel. Though the allocators are completely different, the implementation of canary values and pointer validation would help the zone allocator against tampering by attackers. Kernel Patch Protection would help against the jailbreaking of iOS. Here the kernel is patched in memory, something that patch protection should protect against.

The XNU kernel has with code signing and sandboxing effective measures to protect a system from tampering. Other kernels cannot provide a system with this level of protection. The iOS operating system is hardened is such a way that attackers need multiple vulnerabilities to be able to run custom binaries on the device. This level of hardening makes that the risk of malware infection can be neglected for iOS, something that is of growing concern on the Android platform[1].

If we look at OS X the story is different. OS X is an open platform, much like Windows and Linux. If the user does not have Gatekeeper enabled the attacker can install and run custom binaries. As we presented in Chapter 6, the mitigation techniques can be bypassed by the attacker. In such a situation the quality of the kernel is of a greater importance. The current design of the XNU kernel has not changed since its first release. The micro kernel concept was never actually realized and as a result the Mach and BSD component are still both present in kernel space. Structs have pointers to similar structs in the other component and functionality is implemented twice. This makes an error prone design, regarding locking and overlapping functionality for example. This, combined with the low level of security research on the XNU kernel before the release of iOS, makes that the Windows and Linux kernel are better protected against a direct attack in our opinion. The Windows and Linux kernel are well researched and targeted. Furthermore their design is less prone to errors.

3. Which exploitation techniques can be used under XNU?

Apart from the added complexity due to concurrency and debug possibilities, kernel exploits used to be less complex than user land exploits. For a long time the mitigation techniques that were already common in user land did not exist in kernel space. Addresses were equal across systems and reboots and all memory pages were writable and executable for example. With the current enabled protection mechanisms exploiting the XNU kernel is similar to that of user space, as the protection mechanisms find their origin in user space.

Apart from `NULL` pointer dereference vulnerabilities, no vulnerability classes have been closed by the current mitigation techniques. For all mitigation techniques the attacker must met certain preconditions to bypass them, but gaining arbitrary code execution is possible.

Depending on the target, the attacker might choose to concentrate on specific parts of the kernel. Due to sandboxing a vulnerability that can be triggered from within the sandbox would be preferred, as escaping the sandbox would require at least one extra vulnerability. If code signing is enabled the attacker would benefit from a vulnerability that can be triggered in a few lines. A reference counter overflow vulnerability for example is not ideal on a platform with mandatory code signing.

[1]blog.trendmicro.com/trendlabs-security-intelligence/infographic-behind-the-android-menace-malicious-apps/

4. How can we harden the security of the XNU kernel?

As was shown in figure 8.1, some attack vectors are only protected by kernel ALSR and WˆX. This could be improved by extending the implementation of the current heap and stack protector. Performance is critical in kernel space, something that was taken into account when we formed our recommendations.

The heap protector would benefit from pointer verification with random canaries, which would prevent the attack scenario of an attacker overwriting the next pointer. Listing 23 shows a possible implementation, with a random canary for each zone. For each allocation the two pointers are compared, to detect tampering. Bypassing would require that the attacker is able to read the two pointers, such that he or she can calculate the canary. Preferably the inbound meta data would be removed from the zone and be placed in a look-a-side buffer. This however requires more storage resources and introduces extra complexity to the allocator.

```c
static inline void
free_to_zone(zone_t zone, void *elem) {
    ...
    elem[0] = zone->free_elements;
    elem[1] = zone->free_elements ^ zone->canary;
    ...
}

static inline void
alloc_from_zone(zone_t zone, void **ret) {
    ...
    if (elem[0] ^ zone->canary != elem[1]) {
        panic("corrupted next pointer");
    }
    ...
}
```

Listing 23: adding a canary value

Protecting the next pointer does not protect against the overflowing in an adjacent allocated block. This attack could be partially mitigated by the introduction of red zones between allocations, a shown in figure 8.2. A red zone could be set to 0, which will break all string functions which can mitigate an attack. A red zone is otherwise useful to stop off-by-one vulnerabilities, where the attacker can overflow into the first byte of the adjacent heap block. Depending on the content of the adjacent block, this might allow the attacker to de- or increment an integer or control the lower byte of a pointer.

For heap exploitation precise knowledge of the heap layout is often required. Depending on the vulnerability the attacker needs to know the contents of the adjacent block or the block that is currently on the top of the freelist. The current zone allocator has a deterministic algorithm for handing out free blocks. The allocator can be forced in a predictable state using techniques as Heap Feng Shui. This deterministic property could be altered in a random algorithm which

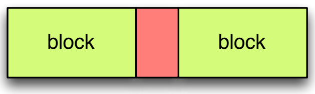


Figure 8.2: a red zone between two allocations

would randomly select a free block upon request, as shown in figure 8.3. Handing out free blocks at random would give the attacker no control over the free list. The attacker might try to use more aggressive heap spraying to bypass this mitigation, but this would increase the chance of a kernel panic.

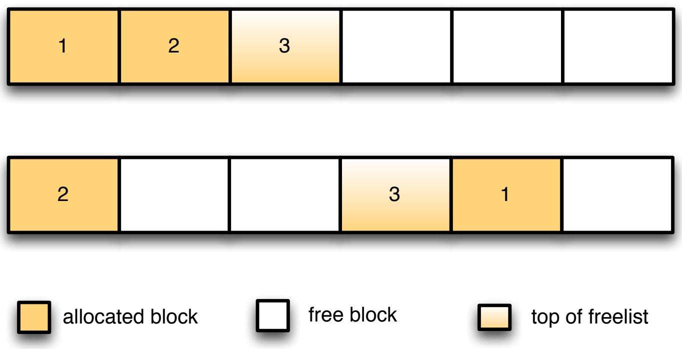


Figure 8.3: random allocations inside a zone

The current stack protector adds a canary value to the stack, to protect stored CPU registers from overwriting. Local variables are normally stored on the stack in the order as they are defined in the source code. This gives the attacker the possibility of overwriting local variables between the buffer and the canary. GCC's stack protector also shuffles local variables such that arrays are stored below other types of local variables, as can be seen in figure 8.4. The compiler of XNU does not reorder local variables at this time. Stack reordering is a compile time option and has no performance impact.

For OS X, support for Intel's SMEP feature could be added. By disallowing the execution of ring 0 code from memory belonging to a higher ring, the attackers are forced into using more complex techniques for gaining code execution.

By extending the security mechanisms as suggested the attack graph from figure 8.1 changes into that of figure 8.5. This graph takes the assumption the attacker can trigger the vulnerability, and is thus not restricted by either code signing or a sandbox.

Even if all suggested mitigation techniques were to be implemented, it would be unable to withstand all attacks. Given an arbitrary read/write primitive for example, the mitigation techniques cannot prevent an attacker from execution arbitrary code. This illustrates the limitations of todays mitigation techniques.

To improve the security of the XNU kernel, and not just mitigate, Apple would

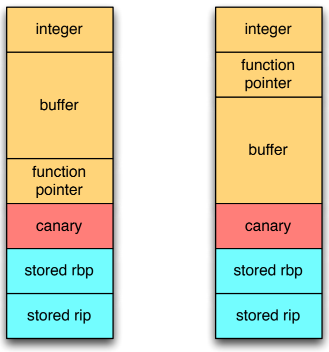


Figure 8.4: a non-reordered stack (left) and a reordered stack (right)

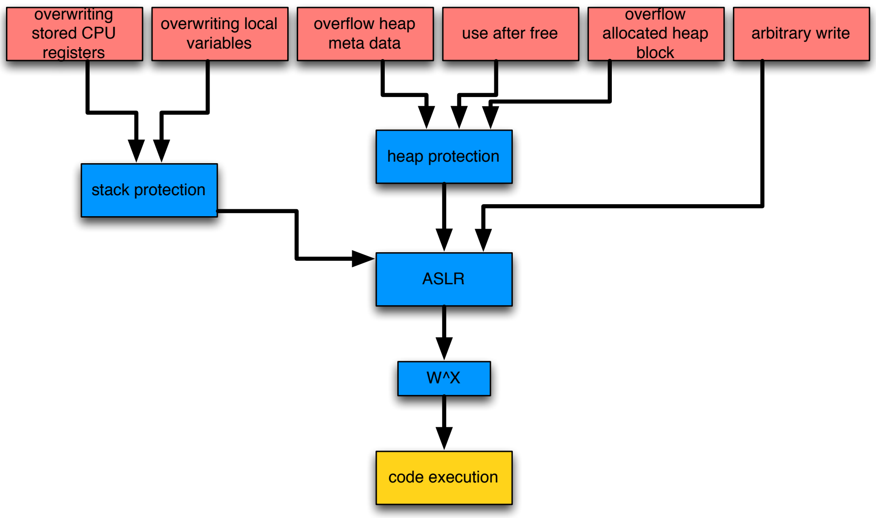


Figure 8.5: situation with extra added protection mechanisms

benefit from a development process that includes security, such as Microsoft's SDL. The latest vulnerabilities in XNU are vulnerabilities that should have been detected (by fuzzing or auditing). This gives the impression that Apple, at least for now, does not maintain a secure development practice.

A final recommendation is the back porting of security fixes to older but still sported releases. For example the vulnerability in listing 24 in `_MALLOC()` is fixed in Mountain Lion since the first beta release (early 2012), but is still present in the latest Lion release (released in September 2012). One can clearly see the integer overflow in the calculation of `memsize`. This makes Mac OS X Lion unnecessary vulnerable, which could be prevented by a single check. Under OS X Mountain Lion the overflow is detected by checking if `size > memsize`, after which `NULL` is returned.

```
bsd/kern/kern_;malloc.c
void *
_MALLOC(
    size_t        size,
    int        type,
    int        flags)
{
    struct _mhead    *hdr;
    size_t        memsize = sizeof (*hdr) + size;

    ...

    if (flags & M_NOWAIT) {
        hdr = (void *)kalloc_noblock(memsize);
    } else {
        hdr = (void *)kalloc(memsize);
        ...
```

Listing 24: an integer overflow in the calculation of `memsize`

A summary of the recommendations is listed below:

- **Memory protection**
  - Improved heap protection:
    - XOR a local copy of the next pointer with a canary value;
    - introduce a red zone between zone blocks to protect against off-by-one vulnerabilities and to stop string functions from overflowing;
    - return a random block on allocation, so that allocations are no longer deterministic.
  - Improved stack protection:
    - reorder local variables, so that stack based buffer overflows are less likely to be exploitable.
  - Implement SMEP support on OS X to prevent execution of kernel space code from a memory page in user space.
- **Secure development practices**
  - Adopt a secure development practice that makes security part of the design process and forces regular security tests.
  - Back port security fixes to older but still supported versions.

8.3 Discussion

Our research shows that Apple is working hard on securing the XNU kernel in the past few years, starting with the release of OS X Lion. Looking at the threats of OS X and iOS it is most likely that Apple tries to protect iOS against jailbreakers. OS X is rarely targeted by attackers and malware either uses social engineering (by disguising themselves as an software update for example) or runs under an unprivileged user account.

XNU is mostly targeted on iOS, where the attacker often already has root privileges. This greatly increases the possible attack surface for the attacker, as he or she is able to access all components of the kernel that are accessible from user space. This attack scenario is different from all other operating systems where the attacker is generally an unprivileged user, who is looking to elevate his or her privileges. The iOS specific attack scenario means that Apple has much more attack surface it has to protect. Code signing is only used since iOS 2.0 (2008), meaning security researchers only just started investigating the security of XNU from a fully privileged user point of view.

The XNU kernel of today still consists out of a separate Mach and BSD component. The design philosophies behind these two kernels are the exact opposites of each other, the one is a full fledged UNIX kernel with processes, terminals and user ID's where the other is a micro kernel using message passing and tasks. Even today, after more then twenty years of development, the two kernels are clearly distinguishable by looking at the source code. This has resulted in many structs that hold pointers to structs used by the other kernel component, as well as functionality that is implemented twice for each component.

Having two separate components in one address space that must heavily interact is an error prone situation. Developers have to pay close attention to locking and the creation and deletion of overlapping data structures for example. It is easy to make a mistake, or having functionality that interferes with functionality that is present in the other component.

As outsiders it is hard to tell if Apple enforces secure development practices of some sort to its developers, but looking at the current state of the XNU kernel we doubt that this is the case. The current state of XNU make it hard to enforce secure design principles, as new features are required to adopt the design choices of XNU, making a secure design for new features impossible.

We see the result in the list of public vulnerabilities, which are often vulnerabilities that are easy to trigger and should normally be found and patched before a public release. As an example we take CVE-2009-1235 in which the kernel incorrectly treats a user provided pointer as a kernel pointer in the `fnctl()` syscall, allowing an attacker to write to arbitrary memory. At the time of discovery XNU did not share its virtual address space with user space (as they do today), meaning the `fnctl()` syscall was never functional. This vulnerability shows that Apple did not, or maybe still doesn't, perform automatic testing on the kernel.

A good example of logic bugs between the Mach and BSD component was presented by Nemo[26], who showed that the root user is able to abuse Mach Traps to write arbitrary kernel memory. This operation is no direct security threat, as the functionality was only accessible for the root user, who could otherwise load a kernel extension with the same results, Though this trick interferes with the secure level protection of the BSD component, which should bring the system in a consistent state (by disallowing the mounting of file systems or the loading of additional kernel extensions for example).

A problem every kernel (or software project) faces is the handling of old, obsolete code. XNU is no exception, as shown by the iOS 5.0.1 jailbreak, which used the obsolete (and non working) `ptrace()` functionality for a sandbox breakout. The sandbox implementation requires that every syscall routine calls the MAC framework, which some syscalls neglect, something that should have been caught in the design stage.

In our opinion the current state of XNU would benefit from the refactoring of the current design. To improve the overall security of the XNU kernel we would recommend to phase out the Mach component of XNU and completely switch to the BSD component, as this components contains most of the features and is a monolithic kernel by design. Furthermore the XNU kernel would benefit from a set of secure development principles, which would include performing regular security tests to eliminate the vulnerabilities that attackers currently use to attack XNU.

A

Toolchain

A.1 Introduction

For security research debugging an application is required to see the current heap layout. This information cannot be obtained from static analysis of the binary. Attaching a debugger at runtime is supported by XNU. We will describe the setup in this chatper.

Large parts of XNU are distributed under an open source license. This is however only true for the OS X version of XNU, for now the iOS version of XNU remains proprietary software. Most parts will be equal across both versions, however there can be slight differences in implementation. Research on closed source components is done by reverse engineering the binary. This chapter will describe the tools that can be used.

The first part of this chapter will describe the steps required for creating a debug environment for the OS X version of XNU. For this a virtual machine is used, which can be controlled by the `gdb` debugger over Ethernet. The second part of this chapter will focus on kernel research for the iOS version of XNU. This section will discuss reverse engineering the kernel binary, as well as enabling kernel debugging using an iPod.

A.2 OS X

The XNU kernels supports a kernel debugging protocol, which allows remote debugging on a running kernel by connecting two Mac's together. With the upcome of different virtualization solutions it is no longer required to have two OS X capable machine's for this, a single OS X capable device with an OS X virtual machine is sufficient. There are multiple applications that support virtualizing OS X, such as VMware Fusion[1] and Parallels Desktop[2] (in our environment VMware Fusion was unable to boot from a debug kernel however). One machine (the virtual machine) is setup to run XNU in debug mode, the host will then attach using Apple's version of the `gdb` debugger. A schematic overview of this setup is shown in figure A.1.

Since the kernel debugging protocol (KDP) uses UDP it should be able to find the connecting `gdb` client. At such low level ARP resolution is unavailable so a

[1]vmware.com/products/fusion
[2]parallels.com/products/desktop

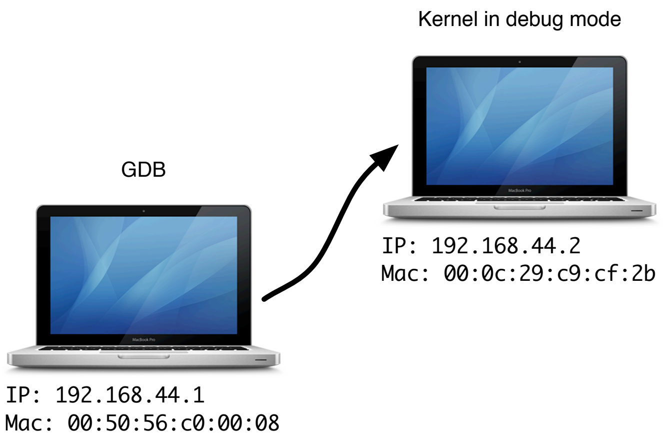


Figure A.1: a remote debugging setup

static ARP entry at both sides is required for the machine's to be able to find each other. Next the kernel should be booted in debug mode, this is done by placing a debug value in nvram. The process for enabling kernel debugging is shown in listing 25. Both the `boot-args` value in nvram and the static ARP entry will survive a reboot.

```
$ sudo arp -s 192.168.44.1 00:50:56:c0:00:08
$ sudo nvram boot-args="-v debug=0x1"
$ sudo reboot
```

Listing 25: enabling kernel debugging

To connect to the debug kernel we need to point `gdb` to the right host and start the debug session. Apple maintains a kernel debugging kit which is available at the developers download page (developer.apple.com/downloads) for registered ADC members (free subscription required). Besides a kernel with debug symbols this also contains a `gdb` script with useful debugging macros. The command `help kgm` gives an overview of the available macros provided by the debugging kit. The process of loading the debug macros and connecting to the debug kernel is shown in listing 26.

In the setup above we set the debug value in nvram to 0x1, which will halt execution at boot and wait for a debugger to attach. After a debugger is attached and execution is continued there is no way to return to the debugging session, since execution will never be halted. A possibility is to set a breakpoint on a rarely used syscall (`fileport_makeport()` for example) and creating a dummy program that will call this function. This will halt execution, allowing the debugger to take control. Waiting on boot for a debugger to attach is the only possibility when the debugging kernel runs inside a virtual machine. If a physical Mac is used, setting the debug value to 0x4 is a better option. This will allow a debugger to attach when a non maskable interrupt is raised (by

```
$ sudo arp -s 192.168.44.2 00:0c:29:c9:cf:2b
$ gdb -q /mach_kernel
(gdb) source kgmacros
(gdb) target remote-kdp
(gdb) attach 192.168.44.2
Kernel is located in memory at 0xffffff801e400000 with uuid \
    of 8D5F8EF3-9D12-384B-8070-EF2A49C45D24
Kernel slid 0x1e200000 in memory.
Connected.
(gdb) break fileport_makeport
Breakpoint 1 at 0xffffff800054cf60
(gdb) continue
```

Listing 26: connecting to a remote kernel

pressing ⌘ – ⏼). Virtualization applications are unable to send a non-maskable interrupt at this time.

Kernel memory can also be made available for user space by enabling the `/dev/kmem` device. This can be done by setting the `kmem` kernel flag in `/Library/Preferences/SystemConfiguration/com.apple.Boot.plist`, as shown in listing 27. After this, kernel memory can be accessed by using `dd` on `/dev/kmem`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD ...">
<plist version="1.0">
<dict>
    <key>Kernel Flags</key>
    <string>kmem=1</string>
</dict>
plist>
```

Listing 27: enabling `/dev/kmem`

A.3 iOS

A.3.1 Extracting kernel images

Firmware files for iOS come as IPSW files, which is actually a ZIP file and can be unzipped using a standard unzipper. Inside an IPSW file there is a file called `kernelcache.release.*`, which is an encrypted IMG3 file containing the kernel image. The image is encrypted using AES and the decryption key and initialization vector are stored within the IMG3 image but encrypted with the device's GID key. The GID key is equal across devices with the same processor class (such as the iPhone 4, iPod 4G and iPad 1). Embedded in the hardware, the GID key cannot be extracted, furthermore access tot the GID key is disabled right before the execution is handed over to the kernel. Meaning that it is only possible to decrypt kernel images from devices that are vulnerable for a boot time vulnerability, so that decrypted images can be extracted before access to the GID key is disabled. At the time of writing there are no known boot time vulnerabilities for the newer devices (iPhone 4S, iPad 2 and newer) and thus firmware images for those devices cannot be decrypted.

The decryption keys for the kernel images can be found on The iPhone Wiki[3] or in the `Key.plist` file included with redsn0w[4]. The iphone-dataprotection project[5] has a tool which is capable of decrypting kernel images by using redsn0w's `Key.plist` file. The steps required for decrypting a kernel image are shown in listing 28.

```
$ hg clone https://code.google.com/p/iphone-dataprotection/
$ cd iphone-dataprotection/
$ sudo port install py27-crypto py27-m2crypto
$ sudo port select --set python python27
$ cp redsn0w.app/Contents/MacOS/Keys.plist .
$ python python_scripts/kernel_patcher.py \
    iPod4,1_5.1.1_9B206_Restore.ipsw
$ file kernelcache.release.n81.patched
kernelcache.release.n81.patched: Mach-O executable arm
```

Listing 28: installing prerequisites and decrypt kernel

A.3.2 Static analysis

Static analysis is a form of reverse engineering, to determine the functionality of an application without running it. This is mostly done by disassembling the application. For disassembling an iOS kernel image the disassembler should support the ARMv7 architecture and the Mach-O file format, for example Hopper Disassembler[6] and Hex-Ray's IDA Pro[7] are capable of this. For IDA Pro a version of 6.2 or higher is recommended, as previous versions are unable to correctly analyzing the kernel image. Older version of IDA Pro mistakenly mark code as data sections and vice versa.

A problem when reverse engineering the iOS kernel is the lack of symbols in the kernel image, as shown in listing 29. The iOS kernel has less than 4 times the number of symbols as the OS X kernel. Without symbols it is difficult to distinguish the different functions and determine their functionality, which can otherwise be done by name. For example all syscall functions have no symbols on iOS.

It is possible to match the symbols of the OS X version of XNU with those of the iOS version. Zynamic's BindDiff[8] is capable of porting symbols across platforms. For syscalls one could also use fingerprinting on the kernel binary

[3]theiphonewiki.com
[4]iphone-dev.org
[5]code.google.com/p/iphone-dataprotection/source/checkout
[6]hopperapp.com
[7]hex-rays.com/products/ida
[8]zynamics.com/bindiff.html

```
$ nm /mach_kernel | wc -l
    16790
$ nm kernelcache.release.n81.decrypted  | wc -l
    4047
```

Listing 29: symbol differences between OS X and iOS

for the `sysent` struct and `mach_trap_table`, which hold a function pointer to all syscalls (this will be described more in depth in chapter 2). The number of arguments for each syscall are also stored in these tables, on a fixed offset from each other which can be used for fingerprinting. A tool that is capable of this is called joker[9], as shown in listing 30.

```
$ ./joker kernelcache.release.n81.decrypted
This is an ARM binary. Applying iOS kernel signatures
Entry point is 0x80085084....This appears to be XNU 2107.2.33
mach_trap_table offset in file/memory (for patching purposes):
 0x2ec4a0/0x802ed4a0
Kern invalid should be 0x80028495. Ignoring those
 10 _kernelrpc_mach_vm_allocate_trap         80014608 T
 12 _kernelrpc_mach_vm_deallocate_trap       80014674 T
 14 _kernelrpc_mach_vm_protect_trap          800146b8 T
 16 _kernelrpc_mach_port_allocate_trap       8001470c T
 17 _kernelrpc_mach_port_destroy_trap        8001475c T
 ...
```

Listing 30: fingerprinting

Static analysis based on its disassembly is still a time consuming task. Decompiling the kernel to pseudo code decreases the time required to determine a methods functionality. Decompilation of the ARM architecture is supported by the Hex-Ray's Decompiler[10], and produces well readable pseudo code in a C like language. Decompilation is especially useful when analyzing large and/or multiple functions at once.

Analyzing the kernel image takes time due to the missing source code and symbols, debugging can safe time in this situation (sometimes this is referred to as dynamic analysis). By attaching a debugger to a running kernel we can break on functions to get a grip on when it gets called and which arguments it receives. For a solid understanding both static analysis and debugging are combined. In the next section we will see how we can attach a debugger to iOS.

A.3.3 Debugging the kernel

[9]newosxbook.com/files/joker
[10]hex-rays.com/products/decompiler

Prerequisites

For attaching a kernel debugger to an iOS device it is required that the device is already jailbroken, since an additional boot parameter is required, as well as a patched kernel. The iOS version of XNU has a function called `PE_i_can_has_debugger` which holds a variable `debug_enable` which should be set to `true` in order to enable debugging. For debugging an iPod or iPad with only Wi-Fi is recommended, since those have no baseband chip. The presence of a baseband chip prevents downgrading of the firmware, as the baseband and the kernel are closely coupled.

Furthermore it should be noted that there is no guarantee that kernel debugging will keep working in later iOS versions. For now the kernel debugging protocol (KDP) is compiled into the iOS version of XNU, but there is no guarantee that Apple will not remove or disable this in a later stage.

The kernel debugging protocol works over UDP and iOS devices do not have an Ethernet interface, some tricks are required to enable kernel debugging. First of all a special cable has to be made that allows kernel debugging over a serial line. Next the iOS kernel should be booted with a special debugging argument such that an OS X machine can connect.For the rest of this section we we used an iPod 4G, however all the steps should also be applicable to other iOS devices.

Building a debug cable

The old iPhone connector is a 30 pins connector, which features not only USB but also a serial connection, which is used for kernel debugging. The new 9 pins Lightning connector, which is used in the new iOS devices, also features a serial connector. We will not cover Lightning in this section due to the lack on a Lightning device. Because Apple does not ship serial capable cables, this cable needs to be custom build as described by Esser[14].

During kernel debugging the Wi-Fi connection can drop. Because we want to control the device over SSH during research it is better is to tunnel SSH traffic over USB, which is more stable. Therefore the debug cable will have two USB connectors, the first connector will connect the iPod over USB, the other will simulate a serial connection. For building the debug cable the following items are required:

- 1 PodBreakout[11]
- 1 Breakout Board for FT232RL USB to Serial[12]
- 2 USB type-A to USB mini type-B cables
- 1 470kΩ resistor

Strip the micro connector of one of the cables, so that the wires can be soldered on the PodBreakout. The resistor has to be soldered on the PodBreakout to enable the serial connection on the iPod. Figure A.2 gives the soldering scheme and figure A.3 shows the final product.

[11]shop.kineteka.com/p/92/
[12]shop.kineteka.com/p/155

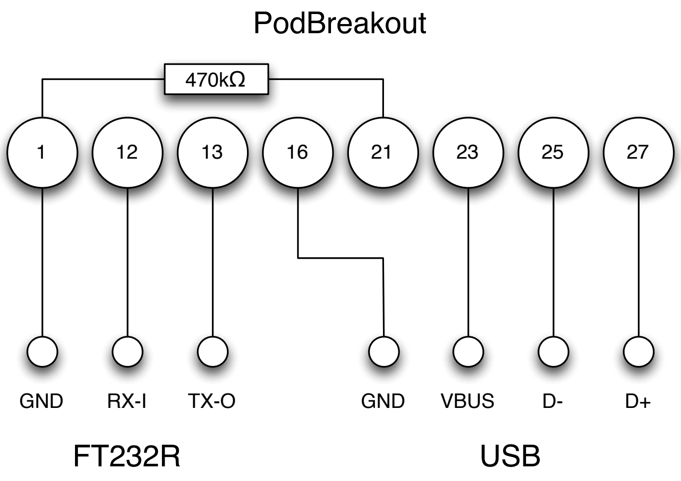


Figure A.2: electrical schema for the debug cable

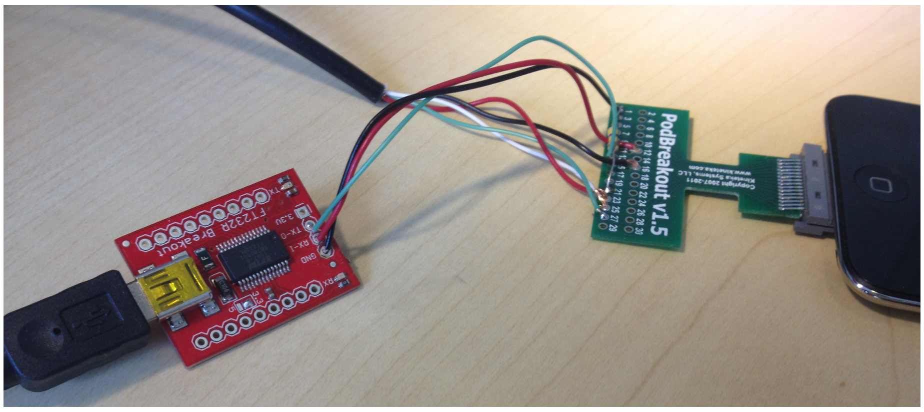


Figure A.3: an iOS debug cable

Setting up iOS

Ideally the device is vulnerable for a Boot ROM exploit. This enables the possibility of passing a boot argument which will allow a debugger to attach. For the newer devices this is not possible so only already jailbroken releases can be debugged by using the jailbreak to alter kernel memory, allowing a debugger to attach after the device has already booted.

Redsn0w is an application used to jailbreak iOS devices. It is build and maintained by the iPhone Dev Team. Besides jailbreaking the application is also capable of passing a boot argument to the iPod (Extras → Even more → Preferences → Boot args). This requires a device that is vulnerable for a Boot ROM exploit (any device with an A4 chip, such as the iPod 4G). Setting the boot argument value to 0x9 will enable kernel debugging (0x1) and print `kprintf()`

output over the serial line (0x8). After setting the boot argument, booting the iPod using redsn0w should make the iPod halt during boot, waiting for a debugger to connect.

Setting up OS X

The FT232RL USB to Serial breakout board requires a driver which can be found at ftdichip.com/Drivers/VCP.htm. This will create a special tty device under `/dev/tty.usbserial*`, which can be used to access the iPod. The kernel debugger on iOS uses the UDP protocol to talk with the attached debugger. Since the iPod is connected via a serial connection, a serial to UDP proxy is required. David Elliott wrote a proxy for this purpose called SerialKDPProxy, which was later updated by Stefan Esser to make it compatible with the later OS X releases. The updated SerialKDPProxy can be downloaded from github.com/stefanesser/serialKDPproxy.

Xcode comes with an ARM capable `gdb` version, which can connect with the iPod. The whole process is listed in listing 31. After the debugger is attached breakpoints can be added and registers and memory can be inspected. Since most symbols are lacking the backtraces will be unreadable.

```
$ SerialKDPProxy /dev/tty.usbserial*
$ /Applications/Xcode.app/Contents/Developer/Platforms \
    /iPhoneOS.platform/Developer/usr/bin/gdb -q        \
     -arch armv7 kernelcache.release.n81.patched
(gdb) target remote-kdp
(gdb) attach 127.0.0.1
Connected.
```

Listing 31: connecting `gdb` to iOS

After the kernel has booted we can login using SSH via USB. For Multiplexing SSH over USB we use a TCP proxy, such as written by the iphone-dataprotection project[13]. This process in shown in listing 32. It requires for an SSH server to be installed on the iPod.

```
$ python python-client/tcprelay.py -t 22:2222
$ ssh -p 2222 root@localhost
```

Listing 32: connecting using SSH over USB

A.4 Conclusion

Due to the fact that the OS X version of XNU is largely open source makes kernel research much easier. The parts that are not open source (most kernel

[13]code.google.com/p/iphone-dataprotection/

extensions are closed source) can be disassembled and decompiled with the same tools as for iOS (Hopper and IDA Pro).

For iOS the research possibilities are much smaller. Bootloader vulnerabilities are patched by Apple. Meaning kernel images can no longer be decrypted, as access to the required GID key is disabled. This makes kernel research on iOS harder as researchers cannot use any form of static of dynamic analysis on the newer iOS kernels.

Now we have created an efficient toolchain for doing kernel research, we dive deeper into the design of the XNU kernel in the next chapter. This describes the core components of XNU and how they interact.

Bibliography

[1] Tarjei Mandt Chris Valasek. Windows 8 heap internals. In *Black Hat USA*, 2012.

[2] Jorge Lucangeli Obes Justin Schuh. A tale of two pwnies. `http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html`.

[3] Deepak Gupta Xiaoning Li. Defeating patchguard. 2012.

[4] Apple. *I/O Kit Fundamentals*, March 2007.

[5] P. Argyroudis and D. Glynos. Protecting the core kernel exploitation mitigations.

[6] ARM. *Procedure Call Standard for ARM Architecture*, October 2009.

[7] ARM. *ARM Architecture Reference Manual*, December 2011.

[8] D. Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *Blackhat, USA*, 2010.

[9] D. Blazakis. The Apple Sandbox. In *Black Hat DC*, 2011.

[10] Andries E. Brouwer. Hackers hut: Exploiting the heap. `http://www.win.tue.nl/~aeb/linux/hh/hh-11.html`.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pages 346–355, 1998.

[12] Bruce Dang. Adventures in analyzing stuxnet. In *27C3*, 2010.

[13] T. de Raadt. Exploit mitigation techniques, 2005.

[14] S. Esser. Exploiting the iOS kernel.

[15] fg! norr. Past and future in os x malware. *Hitcon*, 2012.

[16] Intel. *Intel 64 and IA-32 Architectures Software Develope's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*, March 2012.

[17] K. Johnson and M. Miller. Exploit mitigation improvements in windows 8, 2012.

[18] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, sept. 2005. *Onlin e: http://www. suse. de/˜ krahmer/no-nx. pdf.*

[19] Ivan Krstic. The OS X App Sandbox. In *WWDC*, 2012.

[20] J. Liedtke. Improving IPC by Kernel Design. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 175–188. ACM, 1994.

[21] Lucy. Inside the Mac OS X kernel. In *24C3*, 2007.

[22] Tarjei Mandt. Modern kernel pool exploitation: Attacks and techniques. In *Infiltrate*, 2011.

[23] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System v application binary interface, amd64 architecture processor supplement, 2009.

[24] K. McAllister. Attacking hardened linux systems with kernel jit spraying. `http://mainisusuallyafunction.blogspot.nl/2012/11/attacking-hardened-linux-systems-with.html`.

[25] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.P. Weinmann. *IOS Hacker's Handbook.* Wiley, 2012.

[26] Nemo. Abusing mach on mac os x. *Uniformed vol. 4*, 2006.

[27] Joshua Hill David Wang Nikias Bassen, Cyril. Jailbreak Dream Team. In *Hack in the Box Amsterdam*, 2012.

[28] E. Perla and M. Oldani. *A Guide to Kernel Exploitation: attacking the core.* Syngress, 2010.

[29] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, volume 10, 2003.

[30] D. Rosenberg. Smep: What is it, and how to beat it on linux. `http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/`.

[31] D. Rosenberg. A heap of trouble: Breaking the linux kernel slob allocator. 2012.

[32] Dan Rosenberg and John Oberheide. Stack jacking. In *Hakito Ergo Sum*, 2011.

[33] A. Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[34] G. Taylor and G. Cox. Behind intel's new random-number generator. *IEEE Spectr.*, pages 32–35, 2011.

[35] Ilja van Sprundel. Having fun with Apple's I/O Kit. In *Hack in the Box Amsterdam*, 2010.

[36] J. Viega and G. McGraw. *Building secure software: how to avoid security problems the right way.* Addison-Wesley Professional, 2001.

[37] Jeff Walden. array.join("") is gc-hazardous. `https://bugzilla.mozilla.org/show_bug.cgi?id=720079`.

[38] R. Watson, W. Morrison, C. Vance, and B. Feldman. The trustedbsd mac framework: Extensible kernel access control for freebsd 5.0. In *Proc. 2003 USENIX Annual Technical Conference*, pages 285–296, 2003.

[39] Doctor Web. Doctor web exposes 550 000 strong mac botnet. `http://news.drweb.com/show/?i=2341&lng=en&c=14`.

[40] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, volume 5. San Francisco, CA, 2002.