Protecting the Core
Kernel Exploitation Mitigations

Patroklos Argyroudis, Dimitris Glynos *
Census, Inc.

Abstract

The exploitation of operating system kernel vulnerabilities has received a great deal of attention lately. In userland most generic exploitation approaches have been defeated by countermeasure technologies. Contrary to userland protections, exploitation mitigation mechanisms for kernel memory corruptions have not been widely adopted. Recently this has started to change. Most operating system kernels have started to include countermeasures against NULL page mappings, stack and heap corruptions, as well as for other vulnerability classes. At the same time, researchers have concentrated on developing ways to bypass certain kernel protections on various operating systems. This whitepaper describes the state-of-the-art in kernel exploitation mitigations as adopted by various operating systems (Windows, Linux, Mac OS X, FreeBSD) and mobile platforms (iOS, Android). Moreover, it also provides approaches, hints and references to existing work for bypassing some of these kernel protections.

1 Introduction

The importance of kernel security has become paramount in recent times since operating system kernels have become an attractive target for attackers. The reasons behind this interest lie in the characteristics of modern operating system kernels. Specifically, kernels consist of large code bases with many subsystems. These subsystems interact with each other via complicated interfaces. In addition to the above, operating system kernels have countless entry points for user data. For example, system calls, IOCTLs, filesystems, and network connections, among others, allow user-controllable data to reach important code paths in the kernel. Therefore, there are many possibilities of bugs that can lead to vulnerabilities that compromise the entire system's security model.

The rest of this whitepaper is structured as follows. The next section presents the most common and widely exploited kernel memory corruption vulnerability classes. Section 3 gives an overview of the memory corruption mitigations that have been deployed to address the exploitation of userland vulnerabilities. Section 4 describes the state-of-the-art in kernel exploitation mitigations as adopted by popular operating systems (Windows, Linux, Mac OS X, FreeBSD) and mobile platforms (iOS, Android). In section 5 we provide notes for generic kernel protection bypasses, before we present our conclusions in section 6.

2 Kernel memory corruption vulnerabilities

One of the most common kernel vulnerability classes are NULL pointer dereferences. The value NULL is widely used in kernel code for initialization, to signify the default case, or as a return value on error. In systems where the virtual address space is split into two, one for the kernel and one for the processes, this creates problems when the kernel tries to dereference a NULL pointer. In this case if the user is allowed to map NULL (i.e. the first page which contains the address 0) then he can directly or indirectly control the kernel's code path. More generally, NULL pointer dereference vulnerabilities have clearly demonstrated that there are a lot of problems when the kernel blindly trusts data

*{argp, dimitris} *at* census-labs.com

provided by the userland.

Another class of vulnerabilities that has been exploited in the context of operating system kernels is that of traditional stack overflows. Kernels commonly use two types of stacks. One to service requests from a user process and another one to service requests from kernel internal components. Furthermore, overflows in kernel allocated memory can also be exploited via either corruption of adjacent heap objects or corruption of the memory allocator's metadata.

The implementation bugs that can lead to such memory corruption vulnerabilities are well known and have been analyzed extensively in the relevant published literature. For reasons of completeness we discuss them here briefly. Insufficient validation of user input remains the main bug that leads to kernel memory corruptions. This can either take the form of traditional insufficient bounds checking or user-controlled array indexes and/or reference counters. Signedness bugs can also lead to kernel vulnerabilities. A common case is the one presented in Listing 1.

Listing 1: Example of signedness bug

```
int func(size_t user_size) {
  int size = user_size;
  ...
  if(size < MAX_SIZE) {
  /* do some operation with size
     considered safe */
  }
  ...
}
```

If the unsigned variable user_size is greater than $2^{31} - 1$ then the signed variable size will become negative and the "if" statement will evaluate to true. Although this specific signedness bug is not as common nowadays, it can still be found in certain operating system kernels. Integer overflows can happen as a result of numeric operations, typically multiplications, and can lead the kernel to allocate less memory than required for the target buffer of copied data. Race conditions are particularly applicable to kernels since most CPUs nowadays are SMP. Therefore implementations need to be careful of their use of shared resources to avoid bugs in which a resource changes state between time of validation and time of use.

3 Userland memory corruption mitigations

Most generic userland exploitation approaches have been defeated by countermeasure technologies. In this section we give an overview of these technologies. Stack canaries are probably the most well known and understood exploitation mitigation technology. They are used to protect metadata stored on the stack. These stack metadata (such as the saved return address and the saved frame pointer) can be corrupted via programming bugs, such as stack buffer overflows, and can lead to security vulnerabilities. A similar protection is implemented on the heap with the use of heap canaries. These are guard values used to protect heap management metadata from overflows. Furthermore, heap guard values are also used in some implementations to encode important pointers stored in the metadata. Another heap mitigation is the validation of the destination addresses that metadata pointers of linked lists point to before their corresponding elements are removed from the list (safe unlinking).

Address Space Layout Randomization (ASLR) is an exploitation mitigation that introduces randomness to virtual memory in order to thwart exploitation mechanisms that rely on static addresses. Different operating systems have introduced ASLR for different areas. For example, the location of the userland stack is randomized between application instantiations. Also, the location of the userland heap is randomized. That is, there is a randomized offset added between the base address of the executable and the heap. The base address of dynamic libraries and executables is also randomized on some ASLR implementations.

Several CPUs nowadays provide the feature to forbid execution from memory regions that are marked as non-executable. This technology is known with various names (NX – Non-eXecute, XD – eXecute Disable, XN – eXecute Never, etc.) on various CPU platforms. This feature is used by operating systems to mark regions on the userland stack and heap as non-executable. Mandatory Access Control (MAC) is another defense employed by operating systems to minimize the impact of compromised applications on the system as a whole. Technologies such as SELinux, grsecurity (RBAC)

and AppArmor (path-based) have been developed and, to a certain extend, adopted. A similar goal of containing application compromises is implemented by process debugging protection approaches. These forbid users to debug (their own) processes that are not launched by a debugger.

Compile-time fortifications are also being deployed in the userland of popular operating systems. Features such as -D_FORTIFY_SOURCE=2 and variable reordering are used to harden privileged applications and network daemons. Finally, it must be stressed that grsecurity/PaX [1] is the seminal work on the subject, has influenced most, if not all, of the exploitation mitigation implementations and provides much more defense features than the ones we listed in this section.

4 Kernel exploitation mitigations

4.1 Linux

Our Linux tour on kernel exploitation mitigations will focus on the technologies present in the 2.6.37 release of the kernel. The Linux kernel supports the SSP-type stack protection offered by `GCC`'s `-fstack-protector` option. This protection is enabled by the `CC_STACKPROTECTOR` compile-time option and affects both module and kernel code. This setting also protects any module code that a user might build after kernel installation.

GCC's SSP mechanism implements two types of stack protection, the first one being variable reordering and the second one being a stack canary. Variable reordering makes sure that the compiler places all local array variables in high addresses on the stack, so that an overflow in these arrays may not overwrite other crucial variables such as local function pointers (see Fig.1). In fact, the compiler moves all local function pointers to the lowest addresses of the allocated space in the stack, so that a `memcpy()` on some other allocated variable will not overwrite their values. This type of defense is crucial since it protects the kernel code from attacks that redirect kernel execution by dereferencing local pointers, before the vulnerable function exits.

The most common stack buffer overflow attack involves a local array that has been overflown with bytes that overwrite the saved instruction pointer

memory	stack contents
high address	...
	array variable
↑	integer variable
	function pointer variable
low address	...

Figure 1: Example of local variable placement strategy of GCC SSP

memory	stack contents
high address	...
	function parameters
	saved instruction pointer
↑	saved frame pointer
	canary
	local variables
low address	...

Figure 2: Linux stack canary placement

(or in some cases saved frame pointer). This type of exploitation targets the post-exit conditions of a function. The GCC SSP mechanism protects the kernel code from such exploitation by placing a random "canary" value between the local variables and the saved frame pointer (see Fig.2). At the function epilogue code the kernel checks that the canary value on the stack is still equal to the original canary value. If the two values are found to be different, the kernel panics (see Fig.3).

```
Kernel panic - not syncing: stack-protector:
Kernel stack is corrupted in c10e1ebf

Pid: 9028, comm: canary-test Tainted: G  D  2.6.37 #1
Call Trace:
  [<c1347887>] ? printk+0x18/0x21
  [<c1347761>] panic+0x57/0x165
  [<c1026339>] __stack_chk_fail+0x19/0x30
  [<c10e1ebf>] ? proc_fdinfo_read+0x6f/0x70
  [<c10e1ebf>] proc_fdinfo_read+0x6f/0x70
  [<c10a377d>] ? rw_verify_area+0x5d/0x100
  [<c10a42d9>] vfs_read+0x99/0x140
  [<c10e1e50>] ? proc_fdinfo_read+0x0/0x70
  [<c10a443d>] sys_read+0x3d/0x70
  [<c1002b97>] sysenter_do_call+0x12/0x26
```

Figure 3: Linux kernel panic after detection of a canary overwrite

Listing 2 shows the actual code that handles the canary in the function prologue and epilogue

Listing 2: Canary placement and checking code for Linux

```
prologue: mov %gs:0x14, %edx
          mov %edx, -0x10(%ebp)
          ...
epilogue: mov -0x10(%ebp), %edx
          xor %gs:0x14, %edx
          jne fail
          ...
     fail: call __stack_chk_fail
```

of `proc_fdinfo_read()`. GCC requires the canary value to be located at `%gs:0x14`. The first two instructions (that are part of the function prologue) read the canary from this location and write it on the stack. In the function epilogue, the canary is retrieved from the stack and checked against the value contained in `%gs:0x14`. If the check fails GCC calls the kernel-supplied `__stack_chk_fail` function, which in the case of Linux causes a kernel panic to occur.

At boot time the kernel generates a per-CPU canary value as shown in Listing 3. Afterwards, each time a Lightweight Process (LWP) is created it receives its own random canary value as shown in the code snippets of Listing 4. For the readers that are not acquainted with Linux, a Lightweight Process is the kernel's representation of a scheduled thread. Having per-LWP canaries means that the kernel stack that handles system calls for each thread, receives its own canary. Hence, the kernel side of a multi-threaded userland application will effectively be protected by multiple canaries, one for each application thread.

Since the kernel uses the `-fstack-protector` option of GCC (as opposed to `-fstack-protector-all`), canary protection is offered only for functions that have local character arrays of size 8 or more bytes. Typically, in a kernel image of 16604 functions only 378 (approx. 2%) receive stack canary protection.

The Linux kernel supports multiple types of slab allocators, the most recent of which is the SLUB allocator. A slab allocator is a dynamic memory allocator (i.e. kernel heap allocator) that allocates contiguous "slabs" of memory for object storage purposes. Objects of the same type (i.e. same length) are grouped in "caches" which may span multiple slabs. By having pre-allocated memory for objects of one type, a slab allocator allows the fast creation of new objects of the same type by reclaiming the space of recently "deleted" (or uninitialized) objects.

The SLUB allocator inserts a canary-like region to the end of each allocated object. This word-sized region is called "Red Zone" and contains '0xcc' bytes if the object is in use and '0xbb' bytes if the object is free (i.e. free space from uninitialized or previously deleted object). However, the Red Zone is not a security mechanism to help defend against kernel heap corruptions. It is a tool to help developers identify memory corruption bugs in the kernel code.

To enable the Red Zone feature one must compile the kernel with the `SLUB_DEBUG` option and boot the kernel with the `slub_debug=FZ` parameter. Once a Red Zone overwrite has been detected, the kernel:

1. prints debugging information to the system console (see Fig. 4)
2. restores the contents of the Red Zone
3. continues execution

The Linux kernel also uses some generic memory protection schemes. For instance, it is capable of write-protecting the pages belonging to its own code along with those of read-only data (e.g. built-in firmware, kernel symbol table etc.). For the read-only data pages the execute bit is also disabled on some architectures that support this.

To defend against NULL page (userspace) mappings the kernel enforces a limit on the lowest possible address that a page can be mapped in the address space of a userspace process. The symbol for this threshold is called `mmap_min_addr` and influences any `mmap` call made by a userspace process. By default `mmap_min_addr` defaults to 4096 (i.e. one page over the NULL page on `x86_32`) but the user can modify this in two ways:

- Through a Linux Security Module
- Through a Discretionary Access Control (DAC) mechanism

The SELinux LSM checks the compile-time option `CONFIG_LSM_MMAP_MIN_ADDR` and the permission `MEMPROTECT__MMAP_ZERO` to figure out if

Listing 3: `boot_init_stack_canary` from `arch/x86/include/asm/stackprotector.h`

```
static __always_inline void boot_init_stack_canary(void)
{
  u64 canary;
  u64 tsc;

#ifdef CONFIG_X86_64
  BUILD_BUG_ON(offsetof(union irq_stack_union, stack_canary) != 40);
#endif
  /*
   * We both use the random pool and the current TSC
   * as a source of randomness. The TSC only matters
   * for very early init, there it already has some
   * randomness on most systems. Later on during the
   * bootup the random pool has true entropy too.
   */
  get_random_bytes(&canary, sizeof(canary));
  tsc = __native_read_tsc();
  canary += tsc + (tsc << 32UL);

  current->stack_canary = canary;
#ifdef CONFIG_X86_64
  percpu_write(irq_stack_union.stack_canary, canary);
#else
  percpu_write(stack_canary.canary, canary);
#endif
}
```

Listing 4: per-LWP canary initialization

```
dup_task_struct @ kernel/fork.c:
281: tsk->stack_canary = get_random_int();

drivers/char/random.c:
1627: DEFINE_PER_CPU(__u32 [4], get_random_int_hash);
unsigned int get_random_int(void)
{
  struct keydata *keyptr;
  __u32 *hash = get_cpu_var(get_random_int_hash);
  int ret;

  keyptr = get_keyptr();
  hash[0] += current->pid + jiffies + get_cycles();

  ret = half_md4_transform(hash, keyptr->secret);
  put_cpu_var(get_random_int_hash);

  return ret;
}
```

```
============================================================================
BUG kmalloc-1024: Redzone overwritten
----------------------------------------------------------------------------
INFO: 0xc7ac9018-0xc7ac9018. First byte 0x33 instead of 0xcc
INFO: Slab 0xc7fe5900 objects=15 used=10 fp=0xc7aca850 flags=0x400040c0
INFO: Object 0xc7ac8c18 @offset=3096 fp=0x33333333
Bytes b4 0xc7ac8c08:  00 00 00 00 00 00 00 00 cc cc cc cc 00 00 00 00
  Object 0xc7ac8c18:  33 33 33 33 33 33 33 33 33 33 33 33 33 33 33 33
                                  ...
 Redzone 0xc7ac9018:  33 cc cc cc
 Padding 0xc7ac901c:  00 00 00 00

 Pid: Pid: 8382, comm: cat Not tainted 2.6.37 #2
 Call Trace:
 [<c10a0e77>] print_trailer+0xe7/0x130
 [<c10a152d>] check_bytes_and_report+0xed/0x150
 [<c10a16e0>] check_object+0x150/0x210
 [<c10a1f22>] free_debug_processing+0xd2/0x1b0
 [<c10a35ae>] kfree+0xfe/0x170
 [<c87f31c0>] ? sectest_exploit+0x1a0/0x1ec [sectest_overwrite_slub]
                                  ...
 [<c1002b97>] sysenter_do_call+0x12/0x26
FIX kmalloc-1024: Restoring 0xc7ac9018-0xc7ac9018
```

Figure 4: Linux system console output after detection of a Red Zone overwrite

a user should be able to map pages below `CONFIG_LSM_MMAP_MIN_ADDR`. In the DAC case, the administrator can set the desirable value for `mmap_min_addr` via the `sysctl` command (`sysctl vm.mmap_min_addr`) or the equivalent `/proc` interface (`/proc/sys/vm/mmap_min_addr`).

A case of particular interest to memory mapping attacks is that of "Poison" pointers. The Linux kernel assigns special values known as "Poison values" to members of free'd (or uninitialized) kernel objects. These values help developers distinguish if an invalid memory access is caused by a *use-after-free* kernel bug or not. For example, the linked list API defined in `include/linux/list.h` provides the following two Poison values for `list_item->prev` and `list_item->next` pointers (in non `x86_64` architectures):

- `LIST_POISON1` = 0x00100100
- `LIST_POISON2` = 0x00200200

Unfortunately, if a *use-after-free* bug dereferences these pointers on a `x86_32` system, it will reference a memory location that can be mapped by a userspace process [2]. This would allow an attacker to redirect the kernel execution flow to his own payload in a manner similar to that of a NULL page mapping exploit. Users that wish to protect themselves from this type of threat will want to define an unmappable memory address to the `CONFIG_ILLEGAL_POINTER_DELTA` compile-time option so that the above pointer values are offset by the amount specified by the option.

The Linux kernel supports the dynamic loading of kernel code through Linux Kernel Modules (LKM). To compile a kernel with LKM support one needs to select the `CONFIG_MODULES` configuration option. Once the kernel is running, only a root user or a user with the `CAP_SYS_MODULE` capability will be able to load a module into the kernel. As shown in Fig. 5 modules reside in writable kernel pages.

```
$ cat /proc/modules
sectest 1162 0 [permanent], Live 0xc87f3000
# grep ^0xc87f3000 /debugfs/kernel_page_tables
0xc87f3000-0xc87f4000    4K   RW   GLB   x   pte
```

Figure 5: Example of loaded Linux Kernel Module

The Linux kernel also supports the *auto-loading* of kernel modules, when a user request cannot be fulfilled with the code currently available in the kernel. For example, when the user creates a socket of a specific protocol family, the kernel *auto-loads* code for that family with a call similar to the one shown below:

```
request_module("net-pf-\%d", family);
```

Although this type of demand-loading may be beneficial in terms of user experience, it exposes the kernel to a broader set of security-related dangers. Specifically, it allows an unprivileged user to indirectly load kernel code that might not be of high quality or code that has not been rigorously tested. To make this clearer let us imagine a scenario where an unauthenticated user connects a USB device to a Linux host. The kernel will automatically load the appropriate USB driver and depending on the Desktop Environment used, the kernel might also be forced to load a filesystem driver in order to automatically mount the USB device's filesystem. The attacker could have crafted this filesystem in such a way so that he could trigger a security bug in a beta-quality filesystem driver and eventually gain complete access to the Linux host.

Stock kernels can also be an issue. They usually come with a plethora of modules so that they can be used with a wide variety of software and hardware configurations. As a result of this, they provide the attacker with a much larger attack surface. A recent example of this was the CVE-2010-2959 bug in the Controller Area Network subsystem [3]. The Debian GNU/Linux was one of the many distributions that offered this module in its stock kernel packages. An attacker could easily auto-load the vulnerable module code by creating a CAN socket and then exploit a vulnerability in this code with the proper input. To mitigate this type of attack one has to either manually install only the absolutely necessary modules for the system to function, or to blacklist all unnecessary modules (via `/etc/modprobe.d/blacklist`), or even to disable module loading at compile time. The Linux kernel also supports disabling module loading at runtime. This allows administrators to first load all necessary modules (say, at boot-time) and then disable all further module loading operations. This feature is available through the `/proc` (`/proc/sys/kernel/modules_disabled`) and `sysctl` (`kernel.modules_disabled`) interfaces.

We will close our tour of the Linux kernel security protections by briefly describing the kernel patches provided by the **grsecurity** project [1]. The project offers a hardening solution that strengthens the security of both the kernel and userland.

The PaX patch offers the following kernel protections:

`PAX_KERNEXEC` Non-executable kernel pages through segmentation

`PAX_RANDKSTACK` Randomization of kernel stack address

`PAX_MEMORY_UDEREF` Protection against invalid userland pointer dereferences

`PAX_USERCOPY` Bounds checking on heap objects when copying to/from userland

`PAX_MEMORY_SANITIZE` Sanitization (zero-ing) of freed kernel pages

`PAX_REFCOUNT` Kernel object reference counter overflow protection

The integrated grsecurity patch also contains the following kernel security measures:

`GRKERNSEC_KMEM` Disallow kernel modifications through /dev/mem, /dev/kmem and /dev/port

`GRKERNSEC_IO` Privileged I/O can be disabled (ioperm, iopl)

`GRKERNSEC_VM86` VM86 mode is restricted to users with the `CAP_SYS_RAWIO` capability

`GRKERNSEC_MODHARDEN` Allow module autoloading only for the root user

`GRKERNSEC_HIDESYM, GRKERNSEC_PROC` Non-root users are denied access to kernel symbols and files that reveal sensitive kernel information (see also `GRKERNSEC_PROC_USER` and `GRKERNSEC_PROC_ADD`)

`GRKERNSEC_DMESG` Access to dmesg(8) is forbidden for non-root users

Finally, it should be noted that the grsecurity patches fix the aforementioned Poison pointer attack vector, by supplying "safe" pointer values to `LIST_POISON1` and `LIST_POISON2`, i.e. addresses that cannot be mapped by a userspace process.

4.2 Windows

Windows 7 (NT version 6.1) has a number of kernel exploitation mitigations. The /GS (buffer security check) Visual Studio compiler option can be used when building core kernel components and drivers. In a driver context, on a protected function start a value (cookie) is placed on the stack before the exception handler table and saved registers. On function exit the value is checked to detect stack corruptions. This cookie (guard value) is 32 bits long on 32-bit Windows and 64 bits long (the top 16 bits of which are always clear) on 64-bit Windows. /GS protects functions that have locally declared GS buffers. Specifically, /GS protects functions with a) arrays that are larger than 4 bytes, b) buffers allocated with _alloca, c) data structures of size larger than 8 bytes (with no pointers though) [4].

The initialization of the /GS kernel cookie in a kernel component, for example win32k, is done in the GsDriverEntry function of the component. This is done with the win32k!__security_init_cookie function. At the prologue of a protected function the cookie, which is stored in the global for the kernel component variable win32k!__security_cookie, is XOR'ed with the value of the EBP register and saved on the stack. At the epilogue of the function the cookie is retrieved from its location on the stack and passed to the win32k!__security_check_cookie function. If the cookie has been corrupted the kernel is halted. Fig. 6 illustrates a kernel function's stack protected via /GS.

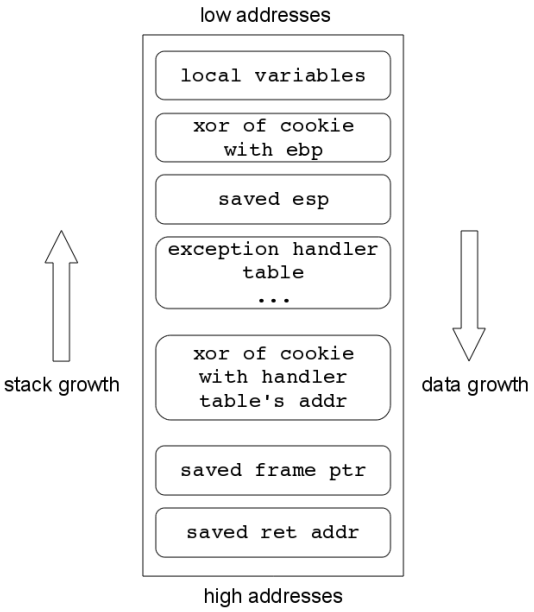

Figure 6: Example of a Windows kernel function's stack protected via /GS

An example success story for the Windows kernel /GS protection is the ICMPv6 router advertisement vulnerability (MS10-009/CVE-2010-0239). This was a remote code execution vulnerability due to unbounded memory copying when processing ICMPv6 router advertisement packets. When the vulnerability is triggered the /GS kernel stack cookie is corrupted and the kernel halts giving a blue screen of death. This is preferable to the alternative which is remote code execution and subsequent full compromise of the system.

There are two ways published in the literature to bypass the /GS kernel stack cookie [2], both of which have specific requirements. The first way is to overwrite the saved return address without corrupting the cookie. This requires to have control over the destination address of the memory corruption bug. The second way consists of overwriting the functions' addresses in the exception handler table stored on the stack. The exception handler table's functions don't need to be in kernel memory and can be overwritten. The requirements for this approach are a) for the exception handler table to exist (i.e. the target driver has registered exceptions), and b) to trigger an exception during or after the kernel stack's corruption.

An approach for guessing the Windows /GS kernel cookie has been recently published [5] and is relevant to local elevation of privilege attacks. The authors discovered that weak entropy sources are used for the generation of the cookie. Specifically, they have found that the cookie for drivers and modules, but not core kernel components (e.g. ntoskrnl.exe etc.), is mainly generated via KeTickCount, i.e. the system tick count value. Based on that they have calculated the prediction success rate at around 46%.

Windows 7 has also introduced safety checks for the kernel's heap allocator to detect corruptions of its metadata. This was introduced to make harder the exploitation of traditional generic unlinking at-

tacks. Such unlinking attacks can be exploited using fake allocator chunks to trigger an arbitrary write-4 primitive. Microsoft's implemented mitigation in Windows 7 is similar to safe unlinking present in other memory allocators. Specifically, the mitigation is the one presented in Listing 5.

The pool implementation of the Windows 7 kernel has been recently attacked with five different methods [6]. In the first one the fact that the safe unlinking implementation does not validate the _LIST_ENTRY of the pool chunk being unlinked, but of the ListHeads the chunk belongs to is exploited. The second attack is about lookaside (single linked) lists in the kernel used for small pool chunks and are not checked for validity. Similarly, the third attack is about PendingFree (single linked) lists used for pool chunks waiting to be freed and are also not checked. In the forth attack exploitation is achieved via the PoolIndex value of the _POOL_DESCRIPTOR structure that is not checked and can be corrupted to point to an attacker-mapped NULL page. The fifth attack is about the corruption of pool chunk pointers to a process object used for reporting usage quota.

NULL page mappings from unprivileged processes are allowed by the Windows 7 kernel, thus making kernel NULL dereferences exploitable. Also, the Windows 7 kernel does not have full ASLR for important kernel structures (e.g.: page tables/directories), but has a kind of *poor man's* ASLR for drivers and nt/hal. The Windows NT kernel (ntkrpamp.exe on SMP+PAE, or generally nt) exports many functions that are useful during the development of kernel mode shellcode. Therefore, the base address of nt needs to be found dynamically before the developed kernel shellcode uses functions of the nt component. One mechanism to achieve this is "scandown" [7].

4.3 Mac OS X

Regarding the Mac OS X operating system we focus on Snow Leopard and specifically version 10.6.6. Snow Leopard by default has a 64-bit userland on a 32-bit kernel, however it can be forced to boot a 64-bit kernel. One of the default security features of Snow Leopard that can hinder some kernel level exploitation approaches is "Secure virtual memory" which is basically encryption for swap. Snow Leopard has no kernel stack smashing protections and no kernel memory allocator protections. However, one of the design choices of the operating system helps in avoiding the exploitation of kernel NULL pointer dereferences. Specifically, Mac OS X has separated kernel and process address spaces, contrary to operating systems that have the kernel mapped at the virtual address space of every process. This means that userland addresses cannot be dereferenced from kernel memory. NULL page mapping from unprivileged processes are allowed but are irrelevant for kernel exploitation purposes. Furthermore, due to this fact userland addresses cannot be used during exploit development for storing fake structures/shellcode/etc.

Snow Leopard also has some minor inconveniences for the attacker. Specifically, the sysent (BSD system call table) symbol is not exported. However, a) it can be dynamically recovered easily and b) the mach_trap_table (Mach system calls) symbol is exported by default. Our tests have also demonstrated that the sysent table is at read-only kernel pages (and the same is true for the kernel pages of the system calls it points to). On the other hand, the mach_trap_table is at writable kernel pages, however the Mach system calls it points to are at pages marked as read-only.

4.4 FreeBSD

The FreeBSD operating system has a number of kernel exploitation mitigations since release 8.0. Kernel stack-smashing protection is implemented via ProPolice/SSP. Specifically, in the sys/kern/stack_protector.c file the functions __stack_chk_init() and __stack_chk_fail() are implemented. The event handler __stack_chk_init() generates a *random* canary value on system *boot*. The canary is placed between a protected kernel function's local variables and saved frame pointer. During the function's epilogue the canary is checked against its original value. If it has been altered the kernel calls __stack_chk_fail() which calls panic(9) bringing down the whole system.

The canary is generated with the arc4rand function which is a random number generator based on the key stream generator of RC4. This is periodically reseeded with entropy from a 256-bit variant of the Yarrow random number generator implemented in the kernel. Yarrow collects entropy from hardware interrupts among other sources. The fact

Listing 5: Windows Safe Unlinking

```
SafeUnlink(Entry)
{
   ...
   Flink = Entry->Flink; // what
   Blink = Entry->Blink; // where
   if (Flink->Blink != Entry) KeBugCheckEx();
   if (Blink->Flink != Entry) KeBugCheckEx();
   Blink->Flink = Flink; // *(where) = what
   Flink->Blink = Blink; // *(what + 4) = where
   ...
}
```

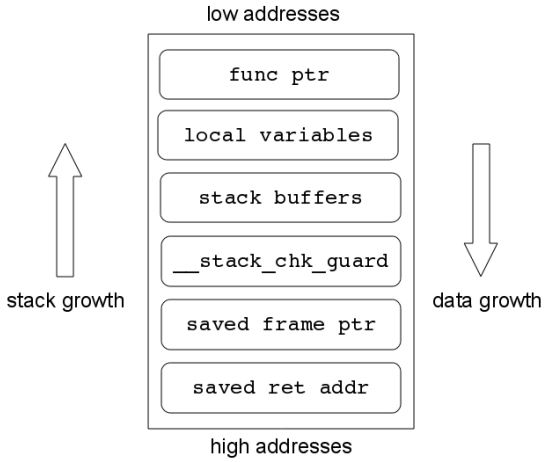

Figure 7: Variable reordering in the FreeBSD kernel by ProPolice/SSP

that FreeBSD's /dev/random never blocks (like Linux's /dev/urandom device) may lead and has led to uniformity flaws [8]. As an example consider a vulnerability published in 2008 which explains how Yarrow provided inadequate entropy to the kernel during boot time [9].

ProPolice/SSP also performs variable reordering. It places local variables below local stack buffers and function pointer arguments below local variables. This means that local variables are placed at *lower addresses* from local stack buffers and function pointer arguments at *lower addresses* from local variables. An example of the above is shown in Fig. 7.

RedZone is oriented more towards debugging FreeBSD's kernel memory allocator (UMA - Universal Memory Allocator) rather than exploitation mitigation. RedZone is disabled by default, the kernel needs to be recompiled with the option DEBUG_REDZONE in order to enable it. RedZone places guard buffers above and below each allocation done via UMA. If the guard buffers are corrupted via an overflow or an underflow then the RedZone implementation calls panic(9). FreeBSD also has protection against unprivileged NULL page mappings. The sysctl(8) variable security.bsd.map_at_zero is enabled by default and forbids mappings of the first page, but allows mappings above that.

4.5 iOS

iOS is Apple's marketing name for the iPhone OS and is directly based on the Mac OS X kernel. It uses a trusted boot process to make sure the firmware has not been altered in any way. Furthermore, it uses code signing of system and application binaries in order to avoid execution of untrusted code. iOS uses sandboxing to limit access to certain parts of the filesystem and system calls. In the userland side of things, iOS has non-executable stack and heap protections enabled by default. However, the absence of ASLR has led to return-oriented-programming (ROP) exploits. The absence of kernel mode protections has led to kernel exploits invoked via ROP sequences to bypass code signing. These kernel exploits were used to make jailbreaks persistent after device reboots by disabling code signing protections [10].

```
$ arm-linux-androideabi-nm libra.ko | \
  grep __stack_chk_fail
$
```

Figure 8: Looking for stack canaries in an Android module

4.6 Android

Google's Android operating system is based on the Linux kernel version 2.6.29 and above. Android targets mainly ARM hardware mobile phone devices, however, the ARM platform's security features are not used by Android. Specifically, the ARM's Digital Rights Management (DRM) implementation called TrustZone is not used in any way [11]. Similarly, ARM's page-level execution protection (XN – eXecute Never) is also not used by Android. In userland, Android applications require permissions for high-level tasks, for example outgoing and incoming network connections. However, once a user has agreed to install an application, thus accepting to grant it all the permissions it requires, this application becomes enabled to execute native code on the device. Subsequently, it can execute kernel exploits to acquire root privileges and completely control the mobile phone. The Android kernel does not use protections available in the mainline Linux kernel that would enable to protect itself from such privilege escalations. For example, the terminal output in Fig. 8 indicates the absence of stack canaries.

5 Bypassing Kernel Protections

There are some generic ways to bypass kernel exploitation mitigations; we discuss them in this section. Canary values on the stack can be found via memory leaks. For the per-LWP canaries on Linux, a same thread leak is required. Ben Hawkes' byte-by-byte canary brute forcing [12] is not applicable in kernel context since kernels panic after a failed guessing attempt. Bypassing NULL page mapping protections requires direct or indirect control of the dereference offset of a kernel pointer. Finally, red zone type kernel heap protections, like those found in Linux and FreeBSD, can be bypassed by overwriting the guards with the right values, which are static and known.

6 Conclusion

In conclusion it can be said that although operating system kernels implement basic proactive security measures, they mostly depend on the quality of their code. Mitigation technologies for kernels will continue to improve albeit slowly since performance impact is a major concern. Despite the available protections the size and complexity of kernels suggests a continuation of exploitable security problems.

Acknowledgments – We would like to thank Matt Miller and Maarten Van Horenbeeck of Microsoft for providing us with helpful comments.

References

[1] "grsecurity kernel patches," http://grsecurity.net.

[2] twiz and sgrakkyu, "A guide to kernel exploitation: attacking the core," http://www.attackingthecore.com/, 2010.

[3] "CVE-2010-2959: Integer overflow in CAN," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2959, 2010.

[4] MSDN, "/GS (Buffer Security Check)," http://msdn.microsoft.com/en-us/library/8dbf701c(v=VS.100).aspx, 2011.

[5] M. Jurczyk and G. Coldwind, "Windows kernel-mode GS cookies subverted," http://j00ru.vexillium.org/?p=690, 2011.

[6] T. Mandt, "Kernel pool exploitation on windows 7," Black Hat DC, 2011.

[7] bugcheck and skape, "Windows kernel-mode payload fundamentals," http://www.uninformed.org/?v=3&a=4&t=txt, 2005.

[8] L. C. Noll, "How good is lavarnd?" http://www.lavarnd.org/what/nist-test.html, 2004.

[9] FreeBSD-SA-08.11.arc4random, "arc4random(9) predictable sequence vulnerability," http://security.freebsd.org/advisories/FreeBSD-SA-08:11.arc4random.asc, 2008.

[10] comex, "Source code of jailbreakme.com," https://github.com/comex/starn, 2010.

[11] J. Oberheide, "Android hax," Summercon, 2010.

[12] B. Hawkes, "Exploiting openbsd," Ruxcon, 2006.