# Protecting the Core
# Kernel Exploitation Mitigations

Patroklos Argyroudis, Dimitris Glynos
{argp, dimitris} *at* census-labs.com

Census, Inc.

Black Hat EU 2011

# Overview

- Importance of Kernel Security
- Kernel memory corruption vulnerabilities
- Userland memory corruption mitigations
- Kernel exploitation mitigations
- Bypassing Kernel Protections
- Conclusion

# Importance of kernel security

- Operating system kernels are an attractive target for attackers
  - Large code bases
  - Countless entry points (syscalls, IOCTLs, FS code, network, etc.)
  - Complicated interactions between subsystems
- Experience has shown that kernels on production systems are seldom upgraded
- Sandbox-based security measures can easily be subverted via kernel vulnerabilities
- Is the requirement of local access relevant anymore?
  - Web apps, devices (iPhone, Android), remote bugs

# Kernel memory corruption vulnerabilities

- NULL pointer dereferences
  - Used for initialization, to signify default, returned on error, etc.
  - Problem for systems that split the virtual address space into two, kernel and process space
- Kernel stack overflows
  - Per-process or per-LWP stacks
  - Kernel internal functions' stacks
- Memory allocator overflows
  - Corrupt adjacent objects
  - Corrupt metadata

# Bugs that lead to memory corruptions

- Insufficient validation of user input
  - Traditional insufficient bounds checking
  - Arbitrary memory corruptions (array indexes, reference counters)
- Signedness

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

- Integer overflows

```
vmalloc(sizeof(struct kvm_cpuid_entry2) * cpuid→nent);
```

- Race conditions
  - Validation time vs use time
  - Changeable locked resources

# USERLAND MEMORY CORRUPTION MITIGATIONS

- Stack canaries
  - Protect metadata stored on the stack
- Heap canaries
  - Guard value
  - Used to encode elements of important structures
- Heap safe unlinking
  - Metadata sanitization
- ASLR
  - Location of stack randomized
  - Random base address for dynamic libraries
  - Random base address for executables (e.g. PIE)
  - Location of heap randomized (e.g. brk ASLR)

# USERLAND MEMORY CORRUPTION MITIGATIONS

- Mark pages as non-executable (DEP/NX/XD/software-enforced)
- Mandatory Access Control (MAC) – SELinux, grsecurity (RBAC), AppArmor (path-based)
- Process debugging protection
  - Forbid users to debug (their own) processes that are not launched by a debugger
  - Contain application compromises
- Compile-time fortification
  - -D_FORTIFY_SOURCE=2
  - Variable reordering
- grsecurity/PaX is the seminal work and provides much more

# Kernel exploitation mitigations

# LINUX

Focus on Linux 2.6.37

- Stack overflow protection
- SLUB Red Zone
- Memory protection
- NULL page mappings
- Poison pointer values
- Linux Kernel Modules
- grsecurity patch

# LINUX :: STACK OVERFLOW PROTECTION

SSP-type protection

- `CC_STACKPROTECTOR` option
- `gcc -fstack-protector`
- affects the compilation of both kernel and modules
- local variable re-ordering
- canary protection only for functions with local character arrays $\geq 8$ bytes
  - in a kernel image with 16604 functions only 378 were protected (about 2%)
  - if the canary is overwritten the kernel panics

# Linux :: Canaries

- A per-CPU canary is generated at boot-time

boot_init_stack_canary @ arch/x86/include/asm/stackprotector.h

```
61 | u64 canary;
62 | u64 tsc;
73 | get_random_bytes(&canary, sizeof(canary));
74 | tsc = __native_read_tsc();
75 | canary += tsc + (tsc << 32UL);
77 | current→stack_canary = canary;
81 | percpu_write(stack_canary.canary, canary);
```

- Each Lightweight Process (LWP) receives its own kernel stack canary

dup_task_struct @ kernel/fork.c

```
281 | tsk→stack_canary = get_random_int()
```

get_random_int @ drivers/char/random.c

```
1634 | hash[0] += current→pid + jiffies + get_cycles();
1635 | ret = half_md4_transform(hash, keyptr→secret);
```

# Linux :: Canaries

- GCC expects to find the canary at %gs:0x14

proc_fdinfo_read @ fs/proc/base.c

```
  9 | mov %gs:0x14, %edx
 16 | mov %edx, -0x10(%ebp)
... | ...
 81 | mov -0x10(%ebp), %edx
 84 | xor %gs:0x14, %edx
 91 | jne <proc_fdinfo_read+106>
... | ...
106 | call <__stack_chk_fail>
```

- The canary is placed right after the local variables, thus “protecting” the saved base pointer, the saved instruction pointer and the function parameters

# Linux :: Stack Overflow Example

```
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
```

# LINUX :: SLUB RED ZONE

- The SLUB is a kernel slab allocator
  - It allocates contiguous "slabs" of memory for object storage
  - Each slab may contain one or more objects
  - Objects are grouped in "caches"
  - Each cache organizes objects of the same type
  - New objects quickly reclaim the space of recently "deleted" objects
- A "Red Zone" is a word-sized canary of '0xcc' bytes placed right after every object in a slab
  - It helps in identifying memory corruption bugs in kernel code (i.e. it's not a security mechanism)
  - If a Red Zone is overwritten, debug info is printed, Red Zone is restored and kernel continues execution
  - Requires `slub_debug=FZ` boot-time option and `SLUB_DEBUG` config option

# Linux :: Slab Overflow Example

```
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
```

# Linux :: Memory Protection

- Right after boot the kernel write protects the pages belonging to:
  - the kernel code
  - the read-only data (built-in firmware, kernel symbol table etc.)
- The non-executable bit is enabled for the pages of read-only data
  - and only on hardware that supports it

# Linux :: NULL Page Mappings

- Linux mmap(2) avoids NULL page mappings by mapping pages at addresses $\geq$ mmap_min_addr
  - mmap_min_addr defaults to 4096
- Two ways to configure mmap_min_addr
  - via a Linux Security Module (LSM)
  - via Discretionary Access Control (DAC)
    - sysctl vm.mmap_min_addr
    - /proc/sys/vm/mmap_min_addr
    - `DEFAULT_MMAP_MIN_ADDR` kernel config option
  - mmap_min_addr $= max(LSM\_value, DAC\_value)$

# Linux :: Poison Pointer Values

- Poison values: special values assigned to members of free'd (or uninitialized) kernel objects
- They help in identifying *use-after-free* bugs
- `LIST_POISON1` and `LIST_POISON2` are Poison values for pointers in linked lists (see `include/linux/list.h`)
- In x86_32 these pointer values default to:
  - `LIST_POISON1` = 0x00100100 **(mappable address!)**
  - `LIST_POISON2` = 0x00200200 **(ditto!)**
- An attacker can exploit a *use-after-free* bug to force the kernel to dereference one of these and ultimately execute his own code found in userspace [ATC2010]
- Mitigation: Provide a safe "base" for these pointers at compile time (see `ILLEGAL_POINTER_VALUE` option)

# LINUX :: KERNEL MODULES

- Kernel code can be loaded at runtime from Linux Kernel Modules (LKM)
- LKM support is configurable at compile time
  - `CONFIG_MODULES` option
- Only root can load a module into the kernel
  - `CAP_SYS_MODULE` capability
- Module code is placed in **writable** pages

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

# Linux :: Kernel Modules

Demand Loading = Trouble!

- Kernel *auto-loads* a (possibly exploitable) module to fulfill a user's request
  - `request_module("net-pf-%d", family);`
- Example #1: Unprivileged user creates socket
  - Kernel loads appropriate module for socket family
- Example #2: Unauthenticated user connects USB storage device
  - Kernel loads appropriate USB driver
  - Desktop environment automatically mounts the device causing a filesystem module to be loaded

# LINUX :: KERNEL MODULES

Demand Loading + Stock Kernels = More Trouble!

- Stock kernels contain modules for all kinds of h/w & s/w configurations
- ...large attack surface that contains code that has not been rigorously tested
- Remember the CAN bug ? (CVE-2010-2959)
  - Debian's stock kernel comes with CAN modules
  - The attacker creates a CAN socket
  - The kernel auto-loads the vulnerable module code
  - The attacker exploits a bug in the CAN code
- Mitigations
  - Install only the modules you need
  - Blacklist unwanted modules
    - /etc/modprobe.d/blacklist
  - Disable module loading (at compile or run time)

# GRSECURITY KERNEL PROTECTIONS

PaX

- `KERNEXEC` – Non-Exec kernel pages (through segmentation)
- `RANDKSTACK` – Randomization of kernel stack
- `MEMORY_UDEREF` – Protection against invalid userland pointer dereference
- `USERCOPY` – Bounds checking on heap objects when copying to/from userland
- `MEMORY_SANITIZE` – Sanitization (zero-ing) of freed kernel pages
- `REFCOUNT` – Kernel object reference counter overflow protection

# GRSECURITY KERNEL PROTECTIONS

Other

- KMEM – No kernel modification via /dev/mem, /dev/kmem, or /dev/port
- IO – Privileged I/O can be disabled (ioperm, iopl)
- VM86 – VM86 mode is restricted (CAP_SYS_RAWIO)
- MODHARDEN – Module auto-loading only for root
- Poison pointer values with safe defaults
- HIDESYM, PROC, PROC_USER, PROC_ADD – Non-root users are denied access to kernel symbols and files that reveal kernel information
- GRKERNSEC_DMESG – Access to dmesg(8) forbidden for non-root users

# WINDOWS

- Focus on Windows 7 (NT 6.1)
  - /GS kernel stack cookie
  - Kernel pool safe unlinking
  - NULL page mappings
  - Kernel ASLR

# WINDOWS :: /GS KERNEL STACK COOKIE

- The /GS (buffer security check) Visual Studio compiler option used when building core kernel components and drivers
- On 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
- 32-bit cookie on 32-bit Windows
- 64-bit (the top 16 bits of which are always clear) on 64-bit Windows

# WINDOWS :: /GS BUFFERS

- Protects functions that have locally declared GS buffers
- Protected:
  - char buf[10];
  - int buf[10]; // only in VS 2010
  - struct { int i; char buf[10]; };
  - struct { int a; int b; int c; int d; }; // only in VS 2010
- Not protected:
  - char buf[4];
  - char *p[10];
  - struct { int i; char *p; };
  - struct { int a; int b; };

# WINDOWS :: /GS COOKIE INITIALIZATION

```
kd> u win32k!GsDriverEntry
win32k!GsDriverEntry:
  8fc73d49 8bff        | mov edi, edi
  8fc73d4b 55          | push ebp
  8fc73d4c 8bec        | mov ebp, esp
  8fc73d4e e8bdffffff  | call win32k!__security_init_cookie

kd> uf win32k!__security_init_cookie+0x12
win32k!__security_init_cookie+0x12:
  8fc73d22 a100f0c38f  | mov eax, dword ptr [win32k!_imp__KeTickCount]
  8fc73d27 8b00        | mov eax, dword ptr [eax]
  8fc73d29 356c63c58f  | xor eax, offset win32k!__security_cookie

kd> dd win32k!__security_cookie
8fc5636c 8fc564ee 703a9b11 00000056 2b7731d5
8fc5637c 4e8bbd79 fcc6da94 180830a1 95baba28
```

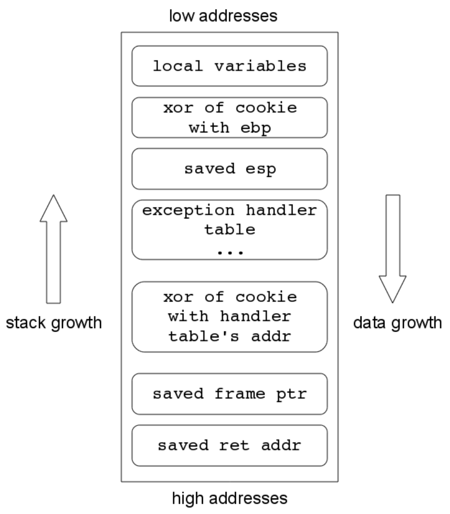
# WINDOWS :: /GS COOKIE USE AND CHECK

```
kd> uf win32k!_SEH_prolog4_GS
win32k!_SEH_prolog4_GS:
  ...                  | ...
  8fb1113d a16c63c58f  | mov eax, dword ptr [win32k!__security_cookie]
  8fb11142 3145fc      | xor dword ptr [ebp-4], eax
  8fb11145 33c5        | xor eax, ebp
  8fb11147 8945e4      | mov dword ptr [ebp-1Ch], eax
```

```
kd> uf win32k!_SEH_epilog4_GS
win32k!_SEH_epilog4_GS:
  ...                  | ...
  8fb11168 8b4de4      | mov ecx, dword ptr [ebp-1Ch]
  8fb1116b 33cd        | xor ecx, ebp
  8fb1116d e846040100  | call win32k!__security_check_cookie
```

```
kd> uf win32k!__security_check_cookie
win32k!__security_check_cookie:
  8fb215b8 3b0d6c63c58f | cmp ecx, dword ptr [win32k!__security_cookie]
  8fb215be 0f85da3f1100 | jne win32k!__report_gsfailure
```

# WINDOWS :: /GS KERNEL STACK PROTECTION



# Ipv6pHandleRouterAdvertisement

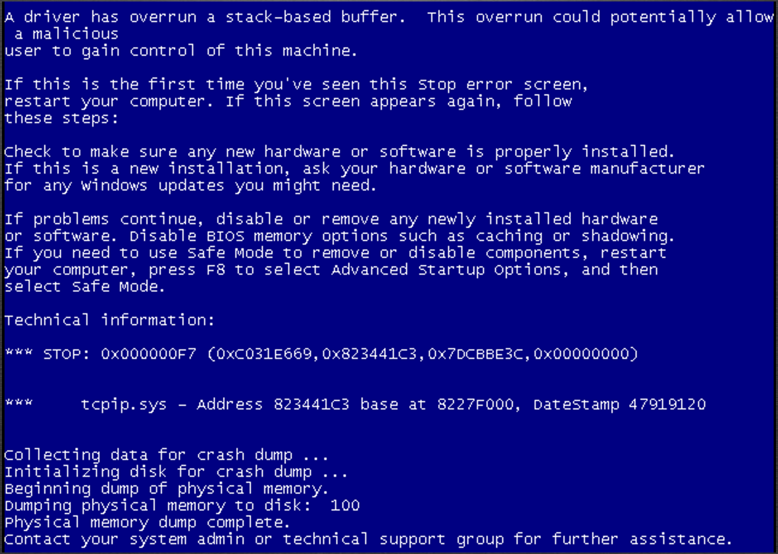
- ICMPv6 router advertisement vulnerability
- MS10-009 / CVE-2010-0239
- Remote code execution vulnerability due to unbounded memory copying when processing ICMPv6 router advertisement packets
- IPv6 enabled by default
- A success story for /GS

# IPV6PHANDLEROUTERADVERTISEMENT

Command - Kernel 'com:port=\\.\com1,resets=0' - WinDbg:6.10.0003.233 X86

```
GSFAILURE_RA_SMASHED:  TRUE

GSFAILURE_ANALYSIS_TEXT: !gs output:
1 Threads detected. Fault occured in thread #0
Corruption occured in tcpip!Ipv6pHandleRouterAdvertisement or one of its callers
Module canary at 0xFFFFFFFF82344004 (tcpip!__security_cookie): 0x823441C3
Complement at 0xFFFFFFFF82344008: 0x7DCBBE3C  (matches OK)

Analyzing __report_gsfailure frame...
LEA usage: Function @0xFFFFFFFF822DDAC4-0xFFFFFFFF822DE3C9 is NOT using LEA
Canary at gsfailure frame not found. (Non-fatal)

Analyzing faulting frame...
Looking for Stack Canary in Function @0xFFFFFFFF822DDAC4 (tcpip!Ipv6pHandleRouterAdvertisement)
Can't find stack canary.
Fatal error - aborting analysis!

Stack buffer overrun analysis completed successfully.


BUGCHECK_STR:  STACK_BUFFER_OVERRUN

SECURITY_COOKIE:  Expected 823441c3 found c031e669

DEFAULT_BUCKET_ID:  VISTA_DRIVER_FAULT

PROCESS_NAME:  System
```

# Ipv6pHandleRouterAdvertisement



# WINDOWS :: BYPASSING KERNEL /GS

- There are two ways published in the literature to bypass the /GS kernel stack cookie [ATC2010]
- Both have requirements

1. Overwrite the saved return address without corrupting the cookie
    - Control the destination address of the memory corruption
2. Exception handler table's functions don't need to be in kernel memory and can be overwritten
    - Exception handler table exists, i.e. the target driver has registered exceptions
    - Trigger an exception during or after the kernel stack's corruption

# Windows :: Guessing the kernel cookie

- Weak entropy sources are used for the /GS kernel cookie generation [JCH2011]
- The cookie is generated once per system session using the following sources for entropy:
  - The address of __security_cookie
  - KeTickCount, i.e. the system tick count value
- A successful prediction consists of calculating
  - the address of __security_cookie,
  - the value of the EBP register,
  - the system tick count
- Authors calculated the prediction success rate at around 46%
- Only applicable to drivers and modules not core kernel components (e.g. ntoskrnl.exe etc.)

# Windows :: Kernel pool safe unlinking

- Safety checks for the kernel's heap allocator to detect corruptions of its metadata
- Introduced to make harder the exploitation of traditional generic unlinking attacks
- Exploitation using fake allocator chunks to trigger an arbitrary write-4 primitive
- Microsoft's implemented mitigation similar to safe unlinking present in other memory allocators

# Windows :: Kernel pool

```
kd> dt nt!_POOL_DESCRIPTOR
    +0x000 PoolType : _POOL_TYPE
    +0x004 PagedLock : _KGUARDED_MUTEX
    +0x004 NonPagedLock : Uint4B
    +0x040 RunningAllocs : Int4B
    +0x044 RunningDeAllocs : Int4B
    +0x048 TotalBigPages : Int4B
    +0x04c ThreadsProcessingDeferrals : Int4B
    +0x050 TotalBytes : Uint4B
    +0x080 PoolIndex : Uint4B
    +0x0c0 TotalPages : Int4B
    +0x100 PendingFrees : Ptr32 Ptr32 Void
    +0x104 PendingFreeDepth : Int4B

    +0x140 ListHeads : [512] _LIST_ENTRY
    // 512 double linked lists that hold free pool chunks
```

# WINDOWS :: LIST ENTRY AND POOL CHUNK HEADER

```
kd> dt nt!_LIST_ENTRY
    +0x000 Flink : Ptr32 _LIST_ENTRY
    +0x004 Blink : Ptr32 _LIST_ENTRY

kd> dt nt!_POOL_HEADER
    +0x000 PreviousSize : Pos 0, 9 Bits
    // BlockSize of previous chunk

    +0x000 PoolIndex : Pos 9, 7 Bits
    +0x002 BlockSize : Pos 0, 9 Bits
    +0x002 PoolType : Pos 9, 7 Bits
    +0x000 Ulong1 : Uint4B
    +0x004 PoolTag : Uint4B
    +0x004 AllocatorBackTraceIndex : Uint2B
    +0x006 PoolTagHash : Uint2B
```

# WINDOWS :: UNLINKING OVERWRITE

```
Unlink(Entry)
{
    ...
    Flink = Entry→Flink; // what
    Blink = Entry→Blink; // where
    Blink→Flink = Flink; // ∗(where) = what
    Flink→Blink = Blink; // ∗(what + 4) = where
    ...
}
```

# WINDOWS :: SAFE UNLINKING

```
ExFreePoolWithTag(Entry, Tag)
{
    if(Entry→BlockSize != Flink→PreviousSize)
        KeBugCheckEx();
}

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
    ...
}
```

# WINDOWS :: OTHER POOL ALLOCATOR ATTACKS

- Five attacks against the latest kernel pool allocator of Windows 7 [KPL2011]
  1. Safe unlinking does not validate the _LIST_ENTRY of the pool chunk being unlinked, but of the ListHeads the chunk belongs to
  2. Lookaside (single linked) lists used for small pool chunks are not checked
  3. PendingFree (single linked) lists used for pool chunks waiting to be freed are not checked
  4. The PoolIndex value of the _POOL_DESCRIPTOR structure is not checked and can be corrupted to point to an attacker mapped NULL page
  5. Pool chunks (optionally) have a pointer to a process object for reporting usage quota

# WINDOWS :: NULL PAGE MAPPINGS

```
DWORD size = 0x1000;
unsigned char payload[] = "\x41\x41\x41\x41 ...";
LPVOID addr = (LPVOID)0x00000004;
// will be rounded to 0x00000000

NtAllocateVirtualMemory(NtCurrentProcess(), &addr, 0, &size,
        MEM_RESERVE | MEM_COMMIT | MEM_TOP_DOWN,
        PAGE_EXECUTE_READWRITE);

memcpy((void *)addr, (void *)payload, sizeof(payload));

kd> u 0
00000000 41        inc ecx
00000001 41        inc ecx
00000002 41        inc ecx
00000003 41        inc ecx
00000004 41        inc ecx
00000005 41        inc ecx
00000006 41        inc ecx
```

# Windows :: Kernel ASLR

- No full ASLR for important kernel structures (e.g.: page tables/directories), but *poor man's* ASLR for drivers and nt/hal
- 6 bits on a 32-bit kernel, 8 bits on a 64-bit kernel
- The Windows NT kernel (ntkrpamp.exe on SMP+PAE, or generally nt) exports many functions
- The base address of nt needs to be found

```
kd> !lmi nt
Module: ntkrpamp
Base Address: 8280d000

kd> !lmi nt
Module: ntkrpamp
Base Address: 8284e000
```

- "Scandown" from a pointer within the nt mapping until the MZ checksum is found [WKP2005]

# Mac OS X

- Focus on Snow Leopard 10.6.6
- By default 64-bit userland on 32-bit kernel
  - Can be forced to boot 64-bit kernel
- Secure virtual memory (i.e. encrypted swap)
- Separated kernel and process address spaces
- No kernel stack smashing protections
- No kernel memory allocator protections
- Some minor *inconveniences* for the attacker

# MAC OS X :: SEPARATED ADDRESS SPACES

- OS X has separated kernel and process address spaces
- Contrary to systems that have the kernel mapped at the virtual address space of every process
- Userland addresses cannot be dereferenced from kernel memory
- NULL page mappings allowed but irrelevant
  - Kernel NULL pointer dereferences become unexploitable
- Cannot use userland addresses during exploit development to store fake structures/shellcode/etc

# Mac OS X :: Minor inconveniences

- The sysent (BSD system call table) symbol is not exported

```
$ nm /mach_kernel | grep sysent
002bf9b0 | T | _hi64_sysenter
0029d7f0 | T | _hi_sysenter
002a0dd0 | T | _lo_sysenter
00831790 | D | _nsysent
0085df9c | S | _nsysent_size_check
0083b140 | D | _systrace_sysent
002a6242 | T | _x86_sysenter_arg_store_isvalid
002a622e | T | _x86_toggle_sysenter_arg_store
```

- The mach_trap_table (Mach system calls) symbol is exported

```
$ nm /mach_kernel | grep mach_trap_table
00801520 | D | _mach_trap_table
```

# Mac OS X :: Writable kernel pages

```
(gdb) p sysent
$6 = {{sy_narg = 0, sy_resv = 0 '\0', sy_flags = 0 '\0',
      sy_call = 0x4954d9 <nosys>,
...
sy_call = 0x483bc4 <getrlimit>, sy_arg_munge32 = 0x4f2d40
      <munge_ww>, sy_arg_munge64 = 0, sy_return_type = 1,
      sy_arg_bytes = 8}, {sy_narg = 2, sy_resv = 0 '\0',
      sy_flags = 0 '\0',
...
(gdb) p getrlimit
$7 = {int (struct proc *, struct getrlimit_args *, int32_t *)}
      0x483bc4 <getrlimit>
(gdb) x/x getrlimit
0x483bc4 <getrlimit>:       0x83e58955
```

# Mac OS X :: Writable kernel pages

```
(gdb) display /i $eip
1: x/i $eip 0x35b146 <tcp_connect+839>:        mov %edx, 0xcc(%edi)
(gdb) set $edx=0x41414141
(gdb) set $edi=getrlimit-0xcc
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0035b146 in tcp_connect (tp=0x483af8, nam=0x21aa3ed8,
        p=<value temporarily unavailable, due to optimizations>)
        at /SourceCache/xnu-1504.9.26/bsd/netinet/tcp_usrreq.c:984
984         tp→cc_send = CC_INC(tcp_ccgen);
1: x/i $eip 0x35b146 <tcp_connect+839>:        mov %edx, 0xcc(%edi)
```

# MAC OS X :: WRITABLE KERNEL PAGES

```
(gdb) display /i $eip
1: x/i $eip 0x35b146 <tcp_connect+839>:     mov %edx, 0xcc(%edi)
(gdb) set $edx=0xcafebabe
(gdb) set $edi=mk_timer_arm_trap-0xcc
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0035b146 in tcp_connect (tp=0x483af8, nam=0x21aa3ed8,
    p=<value temporarily unavailable, due to optimizations>)
    at /SourceCache/xnu-1504.9.26/bsd/netinet/tcp_usrreq.c:984
984     tp→cc_send = CC_INC(tcp_ccgen);
1: x/i $eip 0x35b146 <tcp_connect+839>:     mov %edx, 0xcc(%edi)
```

# Mac OS X :: Writable kernel pages

```
(gdb) display /i $eip
1: x/i $eip 0x35b146 <tcp_connect+839>:       mov %edx, 0xcc(%edi)
(gdb) set $edx=0xcafebabe
(gdb) x/x sysent
0x82eee0 <sysent>:      0x00000000
(gdb) set $edi=0x82eee0-0xcc
(gdb) ni
kdp_reply_wait: error from kdp_receive: receive timeout exceeded
kdp_transaction (kdp_fetch_registers_i386): transaction timed out
```

# MAC OS X :: WRITABLE KERNEL PAGES

```
(gdb) display /i $eip
1: x/i $eip 0x35b146 <tcp_connect+839>:      mov %edx, 0xcc(%edi)
(gdb) set $edx=0xcafebabe
(gdb) x/x mach_trap_table
0x801520 <mach_trap_table>:      0x00000000
(gdb) set $edi=0x801520-0xcc
(gdb) ni
985             if (taop→tao_ccsent != 0 &&
2: x/i $eip 0x35b14c <tcp_connect+845>:      mov 0x4(%eax), %ecx
(gdb) x/x mach_trap_table
0x801520 <mach_trap_table>:      0xcafebabe
```

# FreeBSD

- Focus on version 8.1 (latest stable)
- Kernel ProPolice/SSP
- RedZone
- NULL page mappings
- All introduced in version 8.0

# FreeBSD :: ProPolice/SSP canary

- sys/kern/stack_protector.c implements __stack_chk_init() and __stack_chk_fail()
- Event handler __stack_chk_init() generates a *random* canary value on *boot*
  - Generated with arc4rand()
- Placed between a protected 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)

# FreeBSD :: Canary generation

```
long __stack_chk_guard[8] = {};
...
__stack_chk_init(void *dummy __unused)
{
    ...

    long guard[__arraycount(__stack_chk_guard)];

    arc4rand(guard, sizeof(guard), 0);
    for (i = 0; i ¡ __arraycount(guard); i++)
        __stack_chk_guard[i] = guard[i];
}
```

# FreeBSD :: arc4rand()

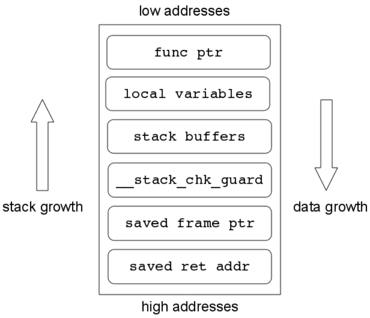
- Random number generator based on the key stream generator of RC4
- Periodically reseeded with entropy from the Yarrow random number generator implemented in the kernel (256-bit variant)
- Yarrow collects entropy from hardware interrupts among other sources
- FreeBSD's /dev/random never blocks (like Linux's /dev/urandom)
  - May lead and has led to uniformity flaws [RND2004]
- Vulnerability in 2008: provided inadequate entropy to the kernel during boot time [FSA2008]

# FreeBSD :: Canary use and check

```
func:         pushl %ebp
func+0x1:     movl %esp, %ebp
func+0x3:     subl $0x210, %esp
func+0x9:     movl 0xc(%ebp), %edx
func+0xc:     movl __stack_chk_guard, %eax
func+0x11:    movl %eax, 0xfffffffc(%ebp)
...
func+0x33:    movl 0xfffffffc(%ebp), %edx
func+0x36:    xorl __stack_chk_guard, %edx
func+0x3c:    jnz func+0x40
func+0x3e:    leave
func+0x3f:    ret
func+0x40:    call __stack_chk_fail
```

# FreeBSD :: Variable reordering

- Local variables placed below local stack buffers
- Function pointer arguments placed below local variables
- That is local variables are placed at *lower addresses* from local stack buffers
- and function pointer arguments are placed at *lower addresses* from local variables



# FreeBSD :: RedZone

- Oriented more towards debugging FreeBSD's kernel memory allocator (UMA - Universal Memory Allocator) rather than exploitation mitigation
- Disabled by default: kernel needs to be recompiled with DEBUG_REDZONE
- Places guard buffers above and below each allocation done via UMA

```
malloc(unsigned long size, struct malloc_type *mtp, int flags)
{ ...
    va = uma_zalloc(zone, flags);
    ...        va = redzone_setup(va, osize);

free(void *addr, struct malloc_type *mtp)
{ ...
    redzone_check(addr);
```

# FreeBSD :: RedZone setup and check

```
redzone_setup(caddr_t raddr, u_long nsize)
{ ...
	haddr = raddr + redzone_roundup(nsize) - REDZONE_HSIZE;
	faddr = haddr + REDZONE_HSIZE + nsize;
	...
	memset(haddr, 0x42, REDZONE_CHSIZE);
	memset(faddr, 0x42, REDZONE_CFSIZE);

redzone_check(caddr_t naddr)
{ ...
	/* Look for buffer overflow. */
	ncorruptions = 0;
	for (i = 0; i < REDZONE_CFSIZE; i++, faddr++) {
		if (*(u_char *)faddr != 0x42)
			ncorruptions++;
```

# FreeBSD :: NULL page mappings

- sysctl(8) variable security.bsd.map_at_zero enabled by default (i.e. the variable has the value 0)

```
void *vptr;
vptr = mmap(0x0, PAGE_SIZE, PROT_READ | PROT_WRITE |
        PROT_EXEC, MAP_ANON | MAP_FIXED, -1, 0);
if(vptr == MAP_FAILED)
{
        perror("mmap");
        exit(EXIT_FAILURE);
}
```

```
$ sysctl -a | grep map_at_zero
security.bsd.map_at_zero: 0
$ ./mmap
mmap: Invalid argument
# sysctl -w security.bsd.map_at_zero=1
$ ./mmap
mmap: 0x0
```

# FreeBSD :: map_at_zero

- From kern/kern_exec.c

```
static int map_at_zero = 0;

int
exec_new_vmspace(image_params *imgp, sysentvec *sv)
{
        ...
        if (map_at_zero)
                sv_minuser = sv→sv_minuser;
        else
                sv_minuser = MAX(sv→sv_minuser, PAGE_SIZE);
```

- Can't map the first page, but can map above that

# iOS

- iOS (Apple's marketing name for the iPhone OS) is directly based on the Mac OS X kernel
- Trusted boot process to make sure the firmware has not been altered
- Code signing of system/application binaries
- Sandboxing to limit access to filesystem/system calls
- Non-executable userland stack and heap
- Absence of ASLR led to return-oriented-programming exploits
- Absence of kernel mode protections led to kernel exploits (invoked via ROP sequences to bypass code signing)
- Executing code in kernel allowed for disabling code signing protections [CSJ2010]

# Android

- Based on Linux kernel 2.6.29+
- ARM hardware devices
- ARM platform's security features not used by Android [HAX2010]
  - TrustZone (Digital Rights Management)
  - XN (eXecute Never) bit page-level protection
- Applications require permissions for high-level tasks
- Native code (i.e. kernel exploits) can be bundled with apps

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

# Bypassing Kernel Protections

- Canary values on the stack can be found via memory leaks
  - For the per-LWP canaries on Linux, a same thread leak is required
- Byte-by-byte canary brute forcing [BHR2006] not applicable in kernel context (kernels panic!)
- Bypassing NULL page mapping protections requires direct or indirect control of the dereference offset of a kernel pointer
- Static red zone type heap protections can be bypassed by overwriting the guards with the right values

# CONCLUSION

- Kernels implement basic proactive security measures
- They mostly depend on the quality of the kernel code :-)
- Mitigation technologies for kernels will continue to improve albeit slowly
  - Performance impact is a major issue
- 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


- [ATC2010] – twiz and sgrakkyu
  A guide to kernel exploitation: attacking the core
  `http://www.attackingthecore.com/`, 2010
- [BSC2010] – MSDN
  /GS (buffer security check)
  `http://msdn.microsoft.com/en-us/library/8dbf701c(v=VS.100).aspx`, 2011
- [JCH2011] – Matthew Jurczyk and Gynvael Coldwind
  Windows kernel-mode GS cookies subverted
  `http://j00ru.vexillium.org/?p=690`, 2011
- [KRH2009] – Larry H.
  Linux kernel heap tampering detection
  *Phrack*, Volume 0x0d, Issue 0x42, 2009
- [KPL2011] – Tarjei Mandt
  Kernel pool exploitation on windows 7
  *Black Hat DC*, 2011

# References


- [WKP2005] – bugcheck and skape
  Windows kernel-mode payload fundamentals
  `http://www.uninformed.org/?v=3&a=4&t=txt`, 2005
- [TBH2010] – Tim Burrell
  The evolution of microsoft's exploit mitigations
  *Hackito Ergo Sum*, 2010
- [RND2004] – Landon Curt Noll
  How good is lavarnd?
  `http://www.lavarnd.org/what/nist-test.html`, 2004
- [FSA2008] – FreeBSD-SA-08.11.arc4random
  arc4random(9) predictable sequence vulnerability
  `http://security.freebsd.org/advisories/FreeBSD-SA-08:11.arc4random.asc`, 2008
- [CSJ2010] – comex
  Source code of jailbreakme.com
  `https://github.com/comex/starn`, 2010

# References


- [GRP2011] – grsecurity/PaX team
  grsecurity/PaX
  `http://grsecurity.net/`, 2011
- [BHR2006] – Ben Hawkes
  Exploiting openbsd
  *Ruxcon*, 2006
- [CVE-2010-2959]
  Integer overflow in CAN
  `http://cve.mitre.org/cvename.cgi?name=CVE-2010-2959`
- [HAX2010] – Jon Oberheide
  Android hax
  *Summercon*, 2010

# QUESTIONS?

Source: Ethan Lofton