A Stitch In Time Saves Nine: A Case Of Multiple OS Vulnerability

Rafal Wojtczuk rafal@bromium.com

Summary

This paper is an analysis of CERT Vulnerability Note VU#649219 , a multiple 64bit OS privilege escalation vulnerability, (re)discovered by the author in April 2012. The nature of the vulnerability and the exploit techniques are covered. A short introduction to x86_64 exception handling is included as a primer in order to clarify the details.

Vulnerability description

CERT Vulnerability Note VU#649219 [1] is titled “SYSRET 64-bit operating system privilege escalation vulnerability on Intel CPU hardware”. While the actual included description is vague, the root cause of the vulnerability is: On some 64bit OSes, untrusted ring3 code can force the kernel to execute *sysret* instruction that would return to a non-canonical address. On Intel CPUs, this results in an exception raised while still in ring0. This exception cannot be handled safely.

The nature of the vulnerability is discussed in more detail further in the paper; here we will focus on the high level overview.

Known vulnerable systems (reminder: only 64bit versions running on Intel CPUs are affected):

- Xen with PV guests
- Windows 7 and Windows 2008 R2
- FreeBSD
- NetBSD

Known non-vulnerable systems

- Apple OSX
- OpenBSD >=5.0 (fixed in July 2011)
- Linux kernel >=2.6.15.5 (fixed in 2006)

While the OpenBSD developers fixed the last remaining exploitation vector accidentally, during code cleanup [2], it seems that Linux developers had a good understanding of the situation. The relevant patch [3] describes explicitly the unexpected environment that the #GP handler executes in upon an exception in *sysret* instruction. However, the author has not found a clear public remark that the

Linux kernel was vulnerable to privilege escalation. The impact is not specified in Linux commits or CVE-2006-0744 [4] description, and the relevant Bugtraq ID 17541 [5] states the issue is DoS only. Also, the aforementioned documents provide no indication that the problem is very low level, and not Linux-specific. This is likely the reason why developers of other operating systems have not noticed the issue, and they remained exploitable for six years.

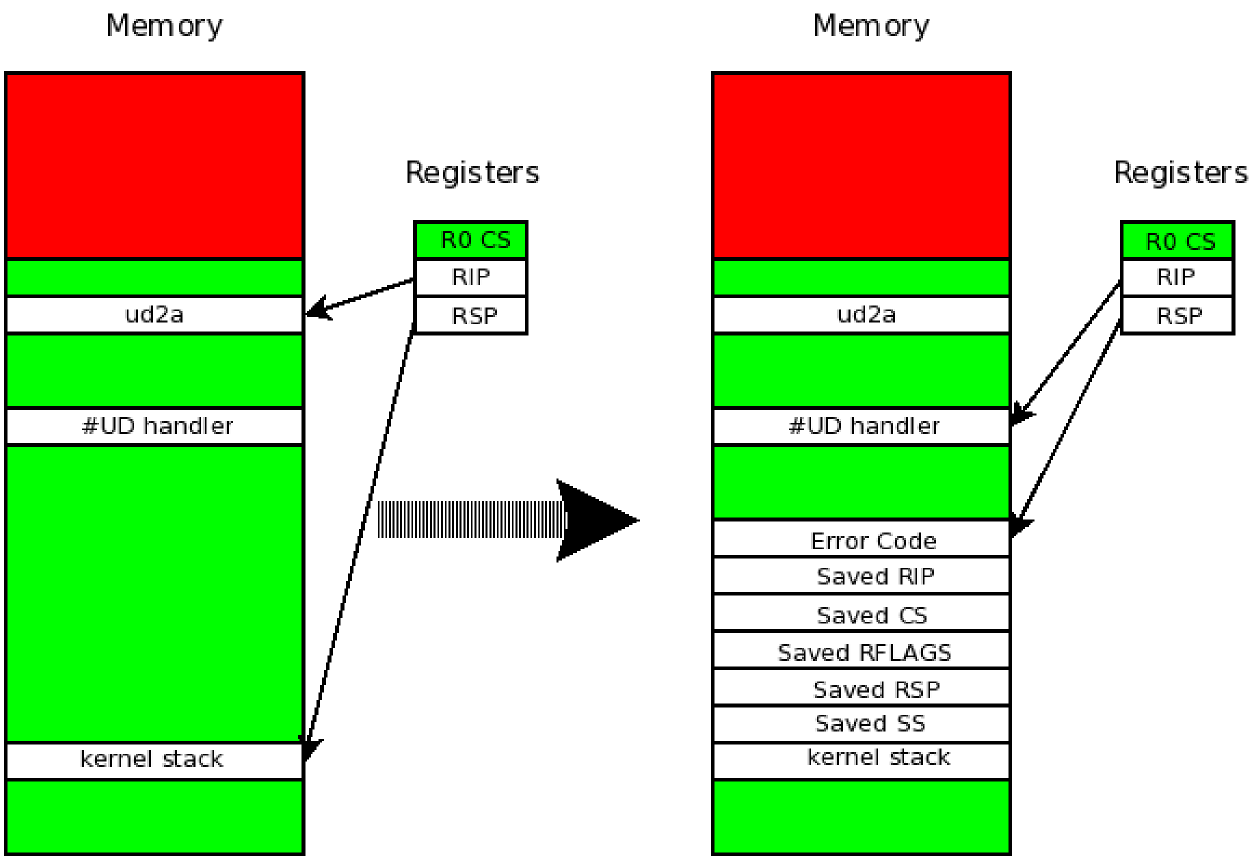
Note: Affected vendors released patches around 12 June 2012.

Crash course on ring transitions on x86_64

In this section, we only outline the most relevant facts, required to understand the nature of the vulnerability. Readers interested with an exhaustive description of exception and syscall handling on x86_64 should resort to Intel's Software Developer's Manual [6].

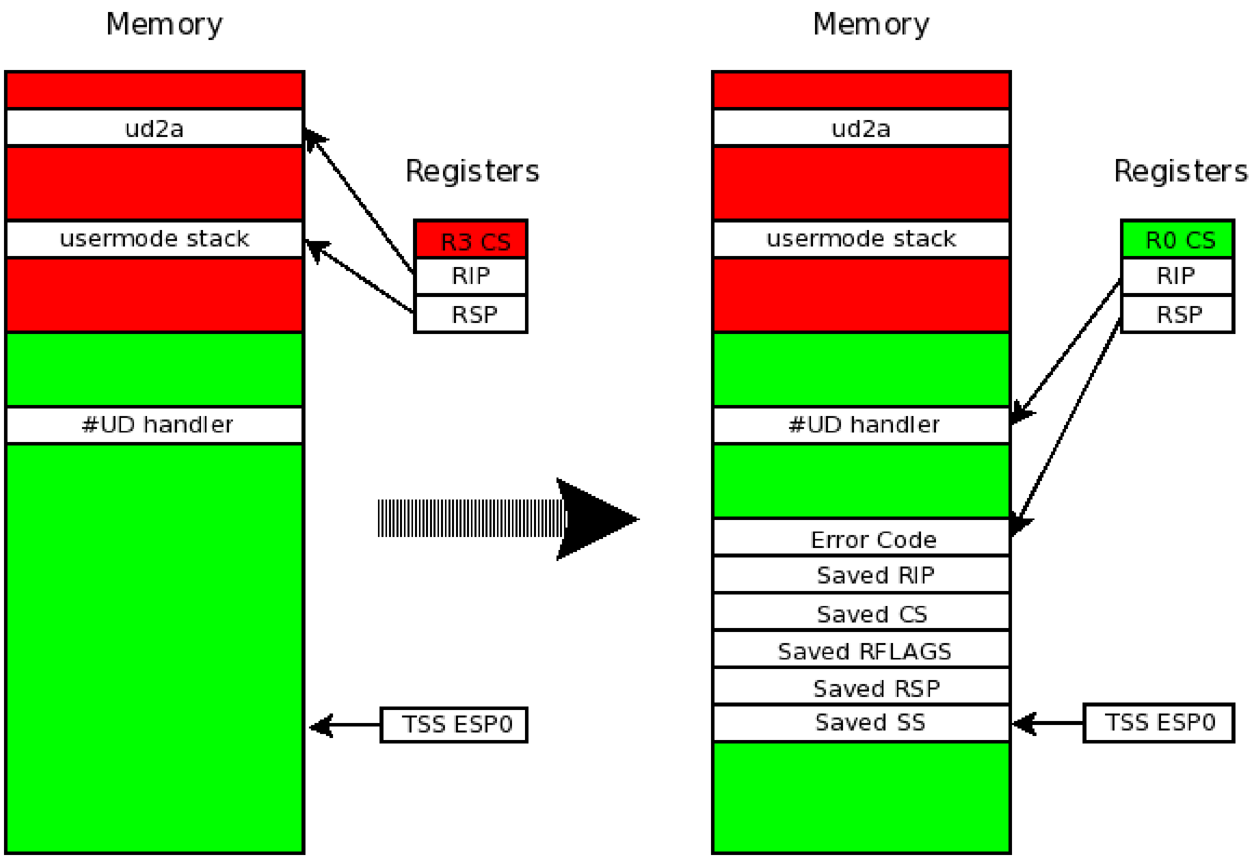
In all the diagrams, the red color means usermode memory, the green color means kernel memory.

When an exception is raised and CPU is in ring 0, the exception record is pushed on the current stack:



More precisely, a mechanism is available (Interrupt Stack Table, IST) that allows the specification of a dedicated stack area even for exceptions triggered in ring 0. However, in compliance with SDM suggestions, all OSes use IST only for catastrophic events (#MC, #DF, #NMI), and all other exceptions behave as described above.

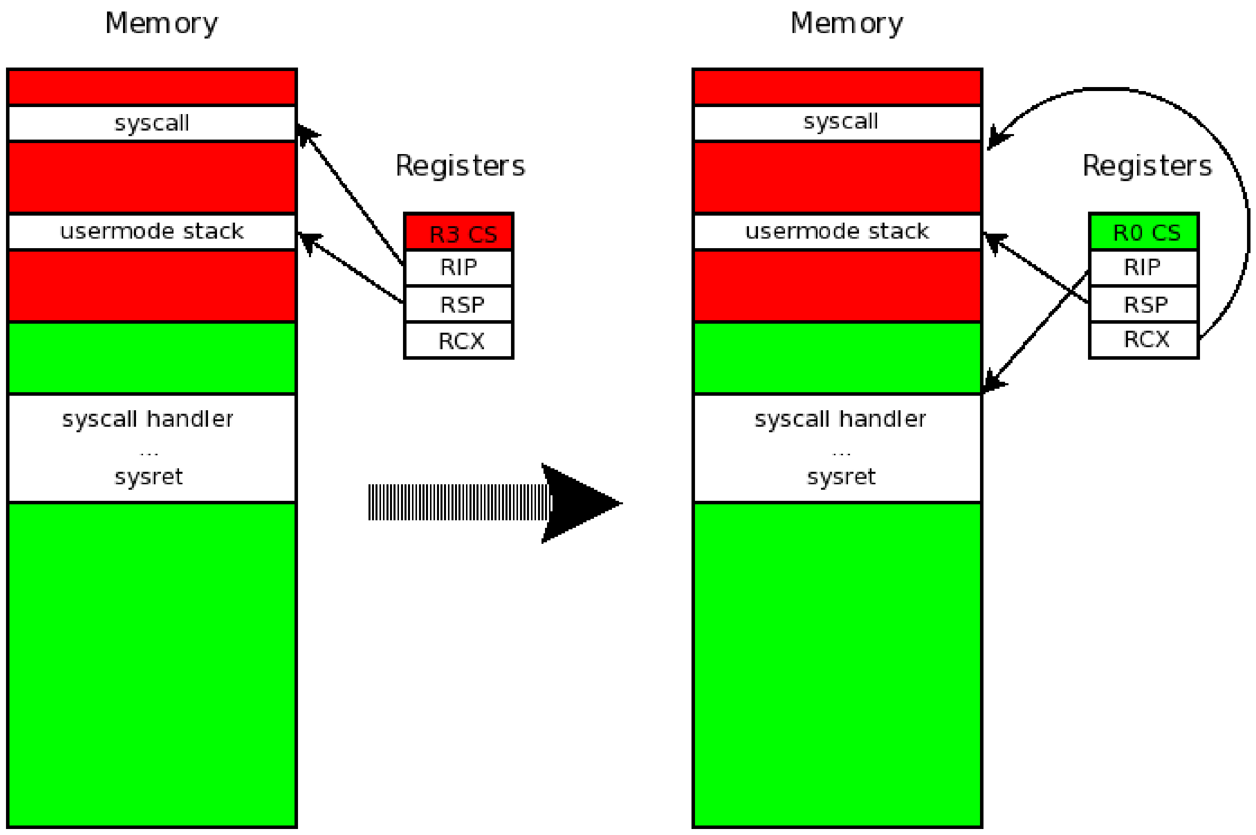
If an exception is raised when CPU is in ring 3, "stack switch" occurs – first, RSP is set to a dedicated kernel stack, and then the exception record is pushed onto it:



Stack switch is a necessary security measure. The most obvious problem is: If the stack pointer is invalid (e.g. points to unmapped memory) at the exception time, and if there was no stack switch, the CPU would attempt to store the exception record in an invalid location. Note that usermode code can set RSP to any value before executing an instruction that throws an exception.

On the other hand, when code is executed in ring 0, the stack pointer is trusted by the CPU to be sane. We will see later that executing an exception handler with a user-controlled stack leads to privilege escalation.

Interestingly, when the *syscall* instruction is executed (which results in transition to ring0 as well), no stack switch occurs:



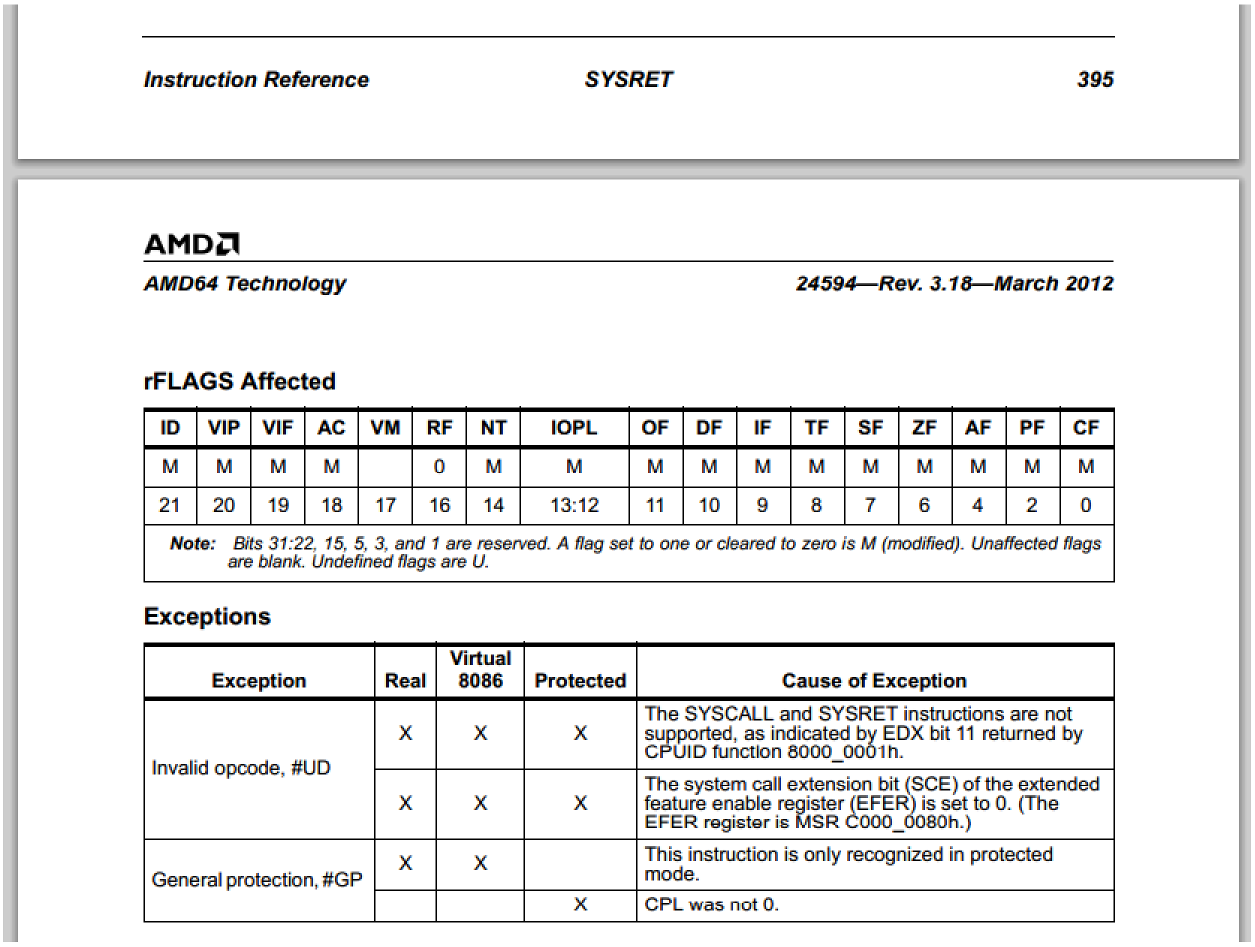
The *syscall* instruction does not use stack at all - it stores the return address, equal to the address of the *syscall* instruction plus two, in RCX register. Still, in order to call normal C code, that actually uses stack for local variables and activation records, the syscall handler must set the stack pointer to a kernel stack. Therefore, the lifecycle of a syscall handler is:

1. Save the usermode-provided RSP somewhere
2. Set RSP to a kernel stack
3. Perform the requested service
4. Set RSP to the saved usermode value
5. Execute the *sysret* instruction

Still, both the prologue and the epilogue of a syscall handler (more precisely, instructions marked as red above) execute in ring 0, with a user-provided, untrusted stack. This means, there must be no hardware interrupt or exception during this time.

The *syscall* instruction does not block interrupts by itself. However, its specification indicates that the RFLAGS register is masked with the IA32_FMASK MSR register. Therefore, all OSes must specify an IF flag in IA32_FMASK MSR in order to prevent interrupts in the syscall handler prologue/epilogue. But that's not enough – the NMI interrupt is not affected by RFLAGS IF bit. Therefore, for security reasons, the NMI handler must use the IST mechanism (note SDM does not mention security as the reason to use IST). Fortunately, it seems all OSes adhere to these requirements.

So, what about exceptions? The assignments to RSP should not normally result in any exception. The only remaining item is the *sysret* instruction itself. The below excerpt from the AMD specification states that on a properly configured system, “sysret” cannot throw any exception:



Instruction Reference *SYSRET* *395*

AMD

AMD64 Technology *24594—Rev. 3.18—March 2012*

rFLAGS Affected

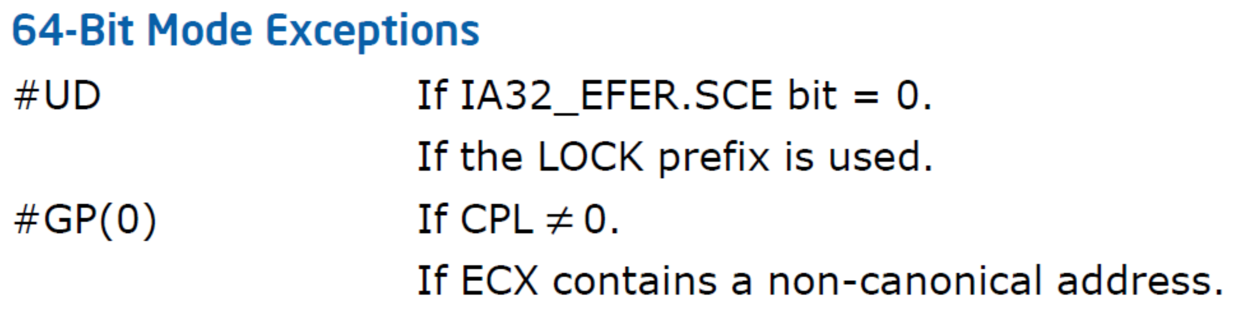
ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
M	M	M	M		0	M	M	M	M	M	M	M	M	M	M	M
21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0

Note: *Bits 31:22, 15, 5, 3, and 1 are reserved. A flag set to one or cleared to zero is M (modified). Unaffected flags are blank. Undefined flags are U.*

Exceptions

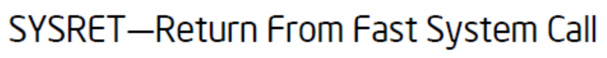
Exception	Real	Virtual 8086	Protected	Cause of Exception
Invalid opcode, #UD	X	X	X	The SYSCALL and SYSRET instructions are not supported, as indicated by EDX bit 11 returned by CPUID function 8000_0001h.
	X	X	X	The system call extension bit (SCE) of the extended feature enable register (EFER) is set to 0. (The EFER register is MSR C000_0080h.)
General protection, #GP	X	X		This instruction is only recognized in protected mode.
			X	CPL was not 0.

Surprisingly, the Intel specification is slightly different:



64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE bit = 0.
	If the LOCK prefix is used.
#GP(0)	If CPL ≠ 0.
	If ECX contains a non-canonical address.



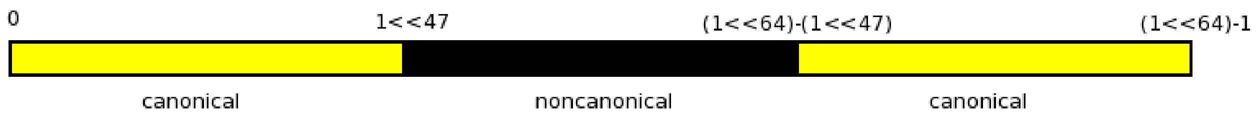
SYSRET—Return From Fast System Call

There is an explicit remark that if the return address (held in RCX) is non-canonical, #GP is raised. As we will cover in the following section, on vulnerable systems such condition can be forced by the malicious usermode code, and as a result, elevation to ring 0 is possible.

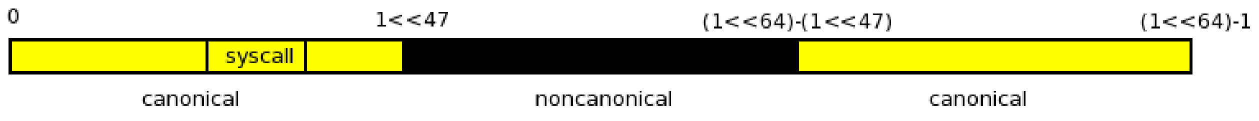
Exploit techniques

Canonical address definition

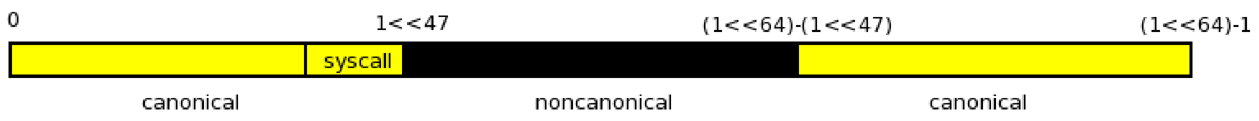
Although on x86_64 systems pointers are 64bit long, supporting full 64bit address space would require many page table levels, which would be inefficient. Instead, only low 48 bits of the pointer can be set arbitrarily and be a valid address. The top 16 bits of a pointer are checked to be the same as the bit 47 – such a pointer is named "canonical", and can be dereferenced. Accessing memory at a non-canonical address results in #GP exception.



Normally, both the address of a *syscall* instruction and the address of the following instruction are canonical:



However, if the "syscall" instruction is placed at virtual address (1<<47)-2, the following address (1<<47) is non-canonical:



Upon returning from a syscall handler, #GP will be raised in ring0.

All kernels place restrictions on the range of virtual addresses that ring 3 code is allowed to map. The following list shows which OSes allow usermode to map a frame at address (1<<47)-PAGE_SIZE:

- Linux – no
- Windows – no
- NetBSD, OpenBSD – no
- FreeBSD – yes
- Xen with PV guests - yes

We will now address all of the exploits, beginning with FreeBSD.

FreeBSD exploit

The DoS exploit is straightforward:

1. Map a frame at (1<<47)-PAGE_SIZE
2. place the *syscall* instruction at (1<<47)-2
3. set RSP to 0 (or any other unmapped address)
4. jump to the syscall instruction at (1<<47)-2

When the syscall handler is terminating with *sysret*, this instruction will raise #GP, which cannot be dispatched because of invalid stack. The result is that the double fault handler invokes successfully, because it is configured to use a dedicated stack via IST mechanism. Usually, #DF handler's purpose is to shut down the system gracefully.

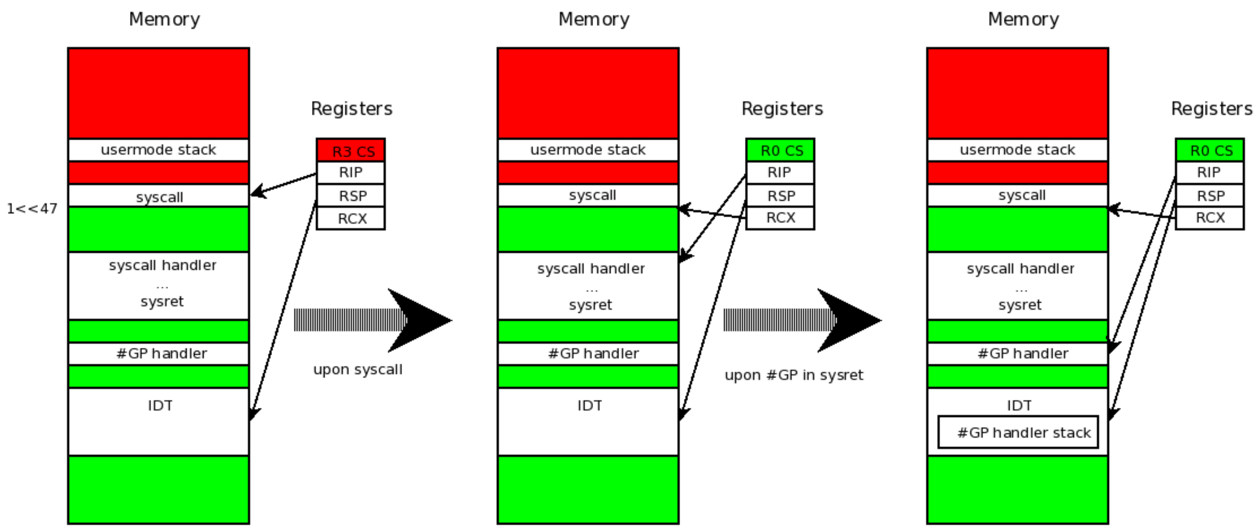
Even though #DF handler may start fine, it could quickly die with a page fault. The reason is usermode *gs* base. See appendix A for a copy of *swapgs* manual from Intel SDM. When *sysret* is being executed, *gs* base has already been reverted to the usermode value. When triggered by ring0 code, exception handlers expect *gs* base to be already switched to kernel address, and do not call *swapgs* in their prologue. Thus when #DF handler tries to access memory with gs: prefix, it will result in page fault (as the default usermode *gs* base is 0 in case of FreeBSD). The resulting triple fault causes instant machine reboot.

What about code execution? The idea is to let #GP handler run, and because there are two execution state elements (RSP and *gs* base) controlled by the user, it is no surprise that code execution will be possible. Because exploiting of *gs* base desynchronization is covered in [7] and [8], we will focus on the possibilities created by controlling RSP.

One possible method is to point RSP to some important kernel data structure. When CPU dispatches #GP, this structure will be overwritten by the exception record. Also, any stack pushes done by #GP handler will overwrite it, too. The natural choice for the overwrite target is IDT, for two reasons:

1. IDT base can be obtained by usermode via the *sidt* instruction. So even when the exploit runs in a restricted environment (say, in a jail) and there is no access to the kernel image to determine the precise kernel binary version, there is no need to guess the location of the overwrite target.
2. Execution needs to be hijacked quickly, before #GP handler shuts down the machine, and before it is terminated by a #PF raise due to memory access with gs: prefix.

Overwriting the #PF handler entry in IDT with an address of kernelmode shellcode works perfectly in FreeBSD case. The following diagram summarizes the exploitation steps:



When #GP handler triggers #PF, the execution will be diverted to the shellcode. The latter has to rebuild the damaged IDT, provide privileges to usermode (by e.g. changing the UID of the current process to 0), and return to usermode via *sysret*.

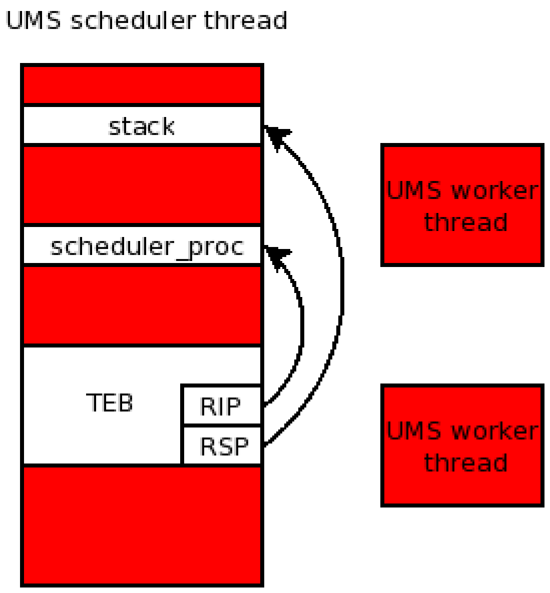
Windows 7 exploit

As mentioned above, on Windows usermode cannot map a page at address (1<<47)-PAGE_SIZE; the usual value of *nt!mmhighestuseraddress* is 0x000007FF`FFFEFFFF.

However, the syscall handler does not always return right after the *syscall* instruction. Jan Beulich spotted it first while developing a patch for Xen for this vulnerability. The primary example is NtContinue system call, used to return from usermode exception handler – it passes full execution context, including RIP, to the kernel, and the latter is supposed to resume execution in the provided context. However, NtContinue uses *iretq* to return to usermode, not *sysret*. Another idea was to change a sleeping thread's RIP via debugger interface – but apparently the newly set RIP is sanitized. Note similar methods may be applicable to Unix systems, *sys_sigreturn* and *ptrace* being equivalents;

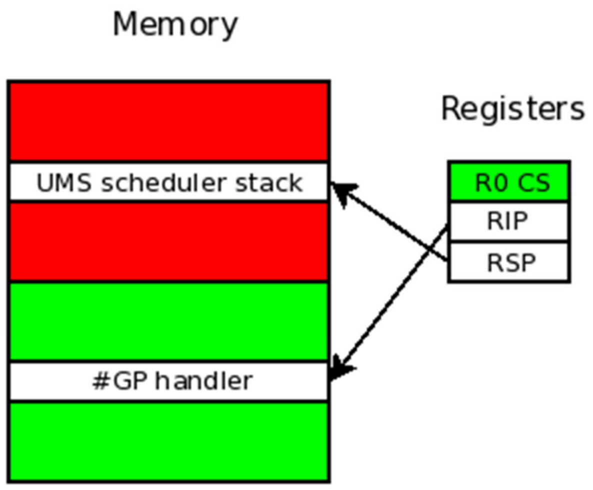
on Unix, *sys_execve* and signal dispatching also return to a RIP unrelated to the address of the previous usermode instruction.

Reverse engineering of *ntoskrnl.exe* revealed one more case when execution is resumed via *sysret* – User Mode Scheduler [9]. The following diagram shows the relevant components:



The function *EnterUmsSchedulingMode* changes the way syscall handler finishes in UMS worker threads. Instead of returning to the calling thread, the context is switched to the UMS scheduler thread, so that the latter can manually schedule an appropriate thread to run. The address of the scheduler procedure (hosted in the UMS scheduler thread), as well as RSP value, is taken from TEB of UMS scheduler thread. It seems to be not sanitized in any way; moreover, *sysret* is used as the method to return to the scheduler procedure. Therefore, if we change the value of scheduler_proc's RIP in TEB to be a non-canonical address, we will arrive at the situation very similar to the one discussed previously: #GP handler will run with user-controlled RSP and *gs* base.

Here is another, more generic exploit scenario. This time, we will let #GP handler run with RSP being set to a mapped usermode address.



Note that usually code treats stack as uninitialized memory. So, it gains nothing to preload the UMS scheduler stack with any payload – code never reads from any location on the stack before writing to it. We need a race condition: we need to poison a crucial stack location after it was initialized by the #GP handler, but before it is read by it. So, concurrently with the UMS scheduler thread, we will run another thread that will continuously write over the stack. We should see the following order of events:

1. #GP handler: calls *some_function*, the saved return address is at stack[*some_offset*]
2. "Overwriter" thread: stack[*some_offset*] := kernel_shellcode
3. *some_function* completes, returns via *ret*
4. control is diverted to kernel_shellcode

This approach looks very promising. "Overwriter" thread's body is only a single *mov* instruction is a loop, so if this thread runs at the moment when *some_function* is called, it should manage to write before *some_function* returns. We can spawn many "overwriter" threads to ensure that at least one of them is scheduled to run at the critical moment. On a common 4-way SMP machine, this method proved to be very effective on a baremetal Windows system.

Somehow unexpectedly, this method appears to be unreliable when Windows is run in a VM. The precise reason is not known – probably, the tested VMM schedules threads in a manner that makes the exploitation fail in many cases. More testing with other VMMs is needed to determine the root cause.

Xen exploit

To change things up - this exploit is very Xen-specific. We will run #GP handler with RSP pointed to usermode stack, but unlike the previous case, without any race condition. It relies on the way Xen obtains the address of the *current* variable, which is a pointer to a crucial data structure (VCPU). Xen expects this pointer to be located below the stack bottom; it even uses the current RSP value to compute *current*'s address. The GET_CURRENT macro expands to:

```
mov $0xffffffffffff8000,%rbx ; set a mask to clear bottom 15 bits

and    %rsp,%rbx   ; get the bottom of per-cpu area

or     $0x7fe8,%rbx ; get the address of "current" pointer

mov    (%rbx),%rbx ;  get the value of "current" pointer
```

This means that although the stack is treated as uninitialized, a crucial location below stack bottom is expected to get initialized. We can poison it with an arbitrary value prior to exploitation, and this value will be used in all operations involving *current* pointer.

Another peculiarity is how Xen's #GP handler determines whether the exception was thrown from usermode. Unlike all other OSes, it does not check the saved CS value in the exception record. It is enough to set bottom 15 bits of RSP to a magic value, and then even though the exception was raised by *sysret* running in ring0, it will be handled as though it was thrown from ring 3. Particularly, #GP handler will try to disassemble the instruction that caused the fault and emulate it. This attempt will fail, and #GP handler will try to inject a #PF fault to the guest. In order to do this, it needs to write to a few fields in the structure pointed to by the *current* pointer. Because we can poison the *current* pointer, it means write-anything-anywhere primitive. It is possible to overwrite the return address of the *do_general_protection()* function with the address of kernelmode shellcode.

Just like all the previous techniques, this method is stable and does not require knowledge of any absolute address of ring 0 data structures – all that is needed is offsets in data structures.

Related remarks

It is very unusual for a single vulnerability to affect so many systems for such a long period of time. Indeed, a stitch in time could have saved nine.

Who failed? Possible explanations:

1. Developers should know the platform they write OS for. *sysret* semantics is explicitly described in Intel SDM. Also, after CVE-2006-0744, everyone should have checked its applicability to their system.
2. It was Intel's mistake to let *sysret* throw an exception. Also, after CVE-2006-0744, Intel should have realized the problem, notified everyone, and updated SDM with an explicit warning.

The "Ivy Bridge" Intel processors, introduced in Q2 2012, have an interesting feature named "Supervisor Mode Execution Prevention" (SMEP). If a SMEP bit in CR4 register is set, then an attempt to execute code stored in usermode page with ring0 CS will fail. Many perceive SMEP as a method to stop common ring0 privilege escalation exploits – indeed, in all the examples above, the ultimate goal was to execute a shellcode located comfortably in user mode pages. The properties of this feature are similar to NX/DEP – if used alone, it can be defeated with return-oriented programming if the addresses of suitable kernel functions are known in advance and the control over stack contents can be gained. It is a step in a right direction, but requires either

1. Making it impossible for an attacker to know the exact kernel version, which is unsuitable for any public vendor
2. ASLR for kernel – not implemented anywhere by now

In both cases, any address leak from the kernel may render the protection useless.

It is unfortunate that current mainline kernel security is so loose among OS vendors. Their code base is huge, which opportunes many vulnerabilities. In general, the separation between kernel and usermode is unsatisfactory – too much state is shared, e.g. virtual memory mapping, which often makes the exploitation reliable. Introduction of SMEP indicates that the need for more strict separation is understood, but it alone is not sufficient. Many have decided to throw the towel on kernel security and use the separation offered by hardware-assisted virtualization, which when used properly, should be more reliable.

Appendix A

"swapgs" instruction description from Intel SDM

SWAPGS exchanges the current GS base register value with the value contained in MSR address 0000102H (MSR_KERNELGSbase). KernelGSbase is guaranteed to be canonical; so SWAPGS does not perform a canonical check. The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel can't save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands.

No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the KernelGSbase MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The KernelGSbase MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. WRMSR will cause a #GP(0) if the value to be written to KernelGSbase MSR is non-canonical.

Bibliography

1. CERT Vulnerability Note VU#649219, http://www.kb.cert.org/vuls/id/649219/
2. Philip Guenther, "Force the sigreturn syscall to return to userspace via iretq", http://www.openbsd.org/cgi-bin/cvsweb/src/sys/arch/amd64/amd64/locore.S.diff?r1=1.47;r2=1.48
3. Andi Kleen, "[PATCH] [18/30] x86_64: When user could have changed RIP always force IRET", http://www.x86-64.org/pipermail/discuss/2006-April/008271.html
4. CVE-2006-0744, http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-0744
5. BID 17541, http://www.securityfocus.com/bid/17541
6. Intel SDM, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
7. Derek Soeder , "VMware Emulation Flaw x64 Guest Privilege Escalation", http://www.securityfocus.com/archive/1/498150
8. Nate Eldredge, "amd64 swapgs local privilege escalation", http://security.freebsd.org/advisories/FreeBSD-SA-08:07.amd64.asc
9. "User Mode Scheduling", http://msdn.microsoft.com/en-us/library/windows/desktop/dd627187(v=vs.85).aspx