OS Security

Memory

Radboud University, Nijmegen, The Netherlands

Winter 2017/2018

Memory access

- So far, all access to resources was handled through file-access permissions
- Requesting a resource (file) is done through syscalls
- Cannot do that for reading/writing memory
- Load/store instructions are very frequent in programs
- Speed of memory access largely determines the speed of many programs
- System calls are expensive
- A load (from cache) can finish in a few cycles
- A system call has some hundred cycles overhead
- OS still needs control over memory access of processes!

Virtual memory

- Central idea:
  - Don't let processes use addresses in physical memory
  - Instead, use *virtual addresses*
  - For each access to a virtual address, map to actual physical address
- Obviously, don't want to map byte-by-byte
- Chop the memory into *pages* of fixed size (typically 4KB)
- Use a *page table* to establish the mapping
- Essentially, use a different page table for each process
- If there is no entry for a virtual address in a processes' page table: exit with segmentation fault

Advantages of virtual memory

- Processes can use (seemingly) contiguous memory locations
- Those addresses don't have to be contiguous in *physical* memory
- Can even assign more memory than is physically available
- Need to swap memory content to and from hard drive
- Can **separate address spaces** of different programs!
- OS can now ensure that one process cannot read/write another processes' memory
- Hmmm, but looking up addresses for each memory access doesn't sound cheaper than a syscall. . .

The MMU

- Mapping from virtual to physical addresses is done in hardware
- CPU has a Memory Management Unit (MMU), which performs the mapping
- Typical setup:
  - OS writes page table for processes to memory
  - OS provides pointer to page table of current process to MMU
  - This is done by writing a special control register, the *page table base register (PTBR)*
  - Access to this control register only from protection ring 0
  - MMU looks up mapping from memory and remembers it in special cache
  - Page-table cache is called *translation lookaside buffer* (TLB)
- Need to invalidate TLB content on context switch:
  - Can flush the whole TLB content
  - Can mark the content invalid and "re-validate" when the process comes back

Shared memory

- Now we have memory of different processes nicely separated
- However, sometimes we want processes to *share memory*
- Shared memory is an efficient (and common) way for inter-process communication (IPC)
- Unix offers syscalls for sharing memory:
  - Can map a file into memory with `mmap()` (with `MAP_SHARED` option)
  - Can request shared memory with `shm_open()` or `shmget()`
  - Shared-memory resources have access permissions similar to files
  - The "execute" flag is ignored
  - For shared memory we're basically back to file access through syscalls

Virtual memory and security

- Virtual memory gives the OS the possibility to *separate memory of different processes*
- One process (or user) can still provide input to another process
- Virtual memory does not say anything about what a process is doing with its own memory!

Why (OS) security fails

- UNIX and Linux assume that user processes behave benignly
- Assumption: **process actions reflect user intentions**
- Mainly two ways why processes may be malicious:
  - user accidently runs malware (more later in the lecture)
  - process operates on maliciously crafted input that exploits **bugs**
- This is a problem of all mainstream “classical” operating systems!
- **Some questions:**
  - Did you ever click on a link to a website?
  - Did you ever open a pdf sent to you by e-mail?
  - Did you ever plug a USB stick into your laptop?
- Ideal situation: OS enforces security:
  - Clearly defined security goals (confidentiality, integrity)
  - All software outside the TCB can be arbitrarily malicious
  - OS still enforces the security goals
- **No current mainstream OS achieves this goal**

Reminder: Memory layout

The memory content of a process is *segmented* into:

- The code segment (or text segment): contains the program code
- The data segment: contains initialized static variables
- The bss segment: contains uninitialized (zeroed) static variables
- The heap: (memory allocated by `malloc` and released by `free`)
- The stack: local data and return addresses
- Memory mapping segment: files, e.g., dynamic libraries mapped into memory

Reminder: The (x86) stack frame

Function call

```
void h() {
  int x = 7;
  int a = 6;
  f(42, 123);
  ...
}

void f(int a, int b) {
char buf[20];
  ...
}
```

Call stack

```
--- stack frame for h ---
7
6
-------------------------
--- stack frame for f ---
123
42
return address to h
frame pointer to h
buf[19]
buf[18]
...
buf[0]
-------------------------
```

A classic buffer-overflow attack

```
#include <stdio.h>

int vulnfunc(void) {
  char *ret;
  char buffer[100];
  ret = gets(buffer);
  printf(buffer);
  printf("\n");
  fflush(stdout);
  if (ret == NULL) return 0;
  else return 1;
}

int main(void) {
  int ret = 1;
  while (ret) {
      ret = vulnfunc();
  }
  return 0;
}
```

- `gets()` reads into `buffer`
- Can write more than 100 bytes to `buffer`
- Fill `buffer` with *shell code*
- Overwrite return address of `vulnfunc()` with address of shell code
- Can write some nops before shell code (“nop slide”)
- Program will jump to shell code and launch a shell

What can the OS do to help?

- Traditional model:
  - User decided to run the program
  - Program behaves benignly (why else would the user run it. . . ?)
  - It's the user's problem
  - Not really helpful with software today
- Two steps to the straight-forward attack:
  1. Change program's control flow
  2. Inject and execute attacker's code
- OS can help to prevent in particular 2
- Compilers can help to prevent 1 (e.g, stack canaries)
- Modern operating systems in fact *do* help

$W \oplus X$

- Real problem of 2. is the von-Neumann architecture
- Code and data share the same memory space
- Idea: Take this back (a little bit)
- Mark some areas of memory (stack, heap, data segment) non-executable
- Such a countermeasure is called *Data Execution Prevention (DEP)*
- Other name: $W \oplus X$ (“either write or execute”)
- Ideally this is implemented in the CPU’s MMU
- Supported by many recent CPUs (e.g., AMD64, ARM)
- Various software solutions for CPUs without hardware support
- Software solutions add overhead to memory access

Enabling/disabling NX

- Non-executable-stack bit is stored in the ELF header of a binary
- Linux by default supports NX stack
- gcc by default produces non-executable-stack binaries
- Disable NX in gcc: `gcc -z execstack`
- Disable NX on an existing binary: `execstack -s BINARY`
- Enable NX on an existing binary: `execstack -c BINARY`
- Disable NX for 32-bit binaries in Linux kernel:
  - Boot parameter `noexec=off` (for x86)
  - Boot parameter `noexec32=off` (for AMD64)
- Reasons to disable NX protection:
  - Creating homework for Software and Websecurity
  - Generally, trying out “classical” attacks
  - Some programs need executable stack!

Return to libc

- Attacker cannot execute his code on the stack anymore
- Workaround: execute code that is already in the program
- (Almost) always mapped into the programs memory space: C standard library
- Idea:
  - Somehow prepare arguments for `system()`
  - overwrite return address with address of `system()`
- Obtain the address of `libc` with
  `cat /proc/$PID/maps | grep libc`
- Obtain the offset of `system()` through

```
nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep system
```

Preparing arguments

- Target: first argument to `system()` should be address of `"/bin/sh"`
- Can write `"/bin/sh"` somewhere
- Alternative: find `"/bin/sh"` somewhere in the binary or libraries
- Then obtain address of `"/bin/sh"`

”The old days” (x86)

- Arguments are passed through the stack
- Write behind buffer
    1. Address of `system()`
    2. Address of `exit()`
    3. Address of `"/bin/sh"`
- Address of `system()` must overwrite return address in current frame
- Code will return to `system()` with
    - return address pointing to `exit()`, and
    - argument pointing to `"/bin/sh"`

Nowadays: AMD64 (x64, x86-64)

- Arguments are passed through registers

Countermeasures

- Can make sure that `\0` is in the address of libc
- Many functions (like `gets`) won't read past the `\0`
- Does not generally help, can overflow some buffers also with `\0`
- Can remove some critical functions from (reduced) libc
- Problems:
    - Can break functionality
    - What functions exactly can cause problems. . . ?

ROP

- We do not have to return to libc functions
- Can also return to arbitrary addresses (e.g., the `pop-retq` gadget)
- Can chain such returns, if each targeted block ends in `return`
- Attack idea: Collect pieces of code from binary (each ending in `return`)
- Chain these pieces to an attack program
- This idea is called *return-oriented programming*
- Concept introduced by Shacham in 2007
- ACM CCS 2017 Test of Time Award
- Collected pieces of code are called *gadgets*
- Attacker now has to program with "gadget-instructions"
- Slight generalization: Can also use gadgets ending in jumps
- Important concept: can obtain *malicious computation* without *malicious code*!
- Searching for gadgets (and to some extent chaining) can be automated

ASLR

- Return to libc and ROP *need to know the addresses of code*
- Idea: randomize position of dynamic libraries
- This approach is called *address space layout randomization (ASLR)*
- Does not only randomize position of dynamic libraries, but also:
  - position of stack
  - position of data segment
  - position of heap
- To also randomize the position of the binary itself need to use `gcc -fpie`
- `pie` stands for "position independent execution"
- Disable ASLR in Linux:
  `echo 0 > /proc/sys/kernel/randomize_va_space`
  or boot with parameter `norandmaps`
- Disable ASLR for one process:
  `setarch ‘uname -m‘ -R PROGRAMNAME`

Attacks against ASLR

- ASLR is generally very effective as a defense
- Problem if address of one instruction leaks to the attacker:
  - Format-string attacks
  - Using overflows to overwrite null-termination
  - Memory content written to disk
  - Software that uses non-randomized modules
  - For a while, `linux-gate.so.1` was not randomized
  - ...
- Problem on 32-bit machines: not enough entropy
  - Cannot randomize lower 12 bits of address (that would break page alignment)
  - Cannot randomize upper 4 bits (limits capabilities of large memory mappings)
  - Result: only 16 bits of entropy (65536 possibilities)
  - Shacham, Page, Pfaff, Goh, Modadugu, Boneh, 2004: brute-force attack that took 216 seconds on average

More things going wrong: race conditions

Definition

A *race condition bug* is a bug where software behaviour depends on uncontrollable timing behavior in an unintended way

An example: TOCTTOU

- Time gap between checking permissions and executing operation
- This is called *time of check to time of use* (TOCTTOU)
- Example: use `access()` syscall in suid-root program to check rights against *real* user ID:

```
if (access("file", W_OK) != 0) {
  exit(1);
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- Attacker attempts to run `symlink("/etc/shadow", "file");` between `access()` and `open()`

A race condition in the Linux kernel

- Announced May 2014: race condition in the Linux kernel
- More specifically: in the `pty` (pseudo-terminal) subsystem
- Bug was there for about 5 years
- Bug allows an attacker to crash the kernel
- Bug allows an attacker to obtain a root shell

The `pty` subsystem

- A `tty` is a typewriter paired with an electromagnetic communication channel
- In the old days: keyboard + screen and OS process (`tty` driver) attached
- Today: Use terminal emulator (e.g., xterm) instead of screen
- Use pseudo-terminal (`pty`) device driver
- Job of the `pty` driver:
  - Read input from one side
  - Parse it for special characters (e.g., CTRL-C, backspace)
  - Handle special characters (e.g, send SIGINT for CTRL-C)
  - Forward the rest to the other side
- Important to notice: Two processes can write to the same `pty`
- Call sequence from userspace program to `pty` buffer:
  `write(pty_fd)` in userspace → `sys_write()` in kernelspace → `tty_write()` → `pty_write()` → `tty_insert_flip_string_fixed_flag()`

The vulnerable code

```
int tty_insert_flip_string_fixed_flag(struct tty_struct *tty,
                                      const unsigned char *chars,
                                      char flag, size_t size)
                                      {
  int copied = 0;
  do {
    int goal = min_t(size_t, size - copied, TTY_BUFFER_PAGE);
    int space = tty_buffer_request_room(tty, goal);
    struct tty_buffer *tb = tty->buf.tail;
    if (unlikely(space == 0))
      break;
    memcpy(tb->char_buf_ptr + tb->used, chars, space);
    memset(tb->flag_buf_ptr + tb->used, flag, space);
    tb->used += space;
    copied += space; chars += space;
  } while (unlikely(size > copied));
  return copied;
}
```

The race condition

Assume two processes write to the same pty

Process A

- `tty_buffer_request_room`
- `memcpy(buf+tb->used,...)`
- `tb->used += space;`

Process B

- `tty_buffer_request_room`
- `memcpy(buf+tb->used,...)`

- `memcpy`(s) of A fill the buffer(s) and increase `used`
- `memcpy`(s) of B will write behind the buffer
- Local-root exploit needs some more bits and pieces, for details see http://blog.includesecurity.com/2014/06/exploit-walkthrough-cve-2014-0196-pty-kernel-race-condition.html