Buffer Overflows
• General problem: **unsanitized user input**

• Low level language (eg C): **overflow a local array (buffer)**

• Write over the **stack**!

• Overwrite the return address

• Execute **adversary-controlled code**
  - from the target program, a library, etc
  - or stored in the buffer
It’s much easier to understand buffer overflows by reproducing one.

Try to reproduce the one we live-coded in the lecture.
   - Use the given code & Makefile

The slides will guide you through the process.

Read also while progressing:
   - Aleph One, *Smashing The Stack For Fun And Profit*
Outline

- Understand the stack
- Disassemble a test program
- Produce an overflow, watch the return address being overwritten
- Write a shellcode in C
- Write a shellcode in assembly, obtain machine code
- Test the binary, overflow our own buffer
- 1st attack: guess the buffer’s address in the target
- 2nd attack: add NOPs for faster guessing
The stack

- Grows with every function call (towards lower addresses)

**Caller**
- stores function arguments in reverse order
- makes call, which stores EIP (return addr.)

**Callee**
- saves old EBP, sets EBP = ESP
- lowers ESP to make room for local vars (also saves some registers, if needed)
- Args: EBP+n
- Local vars: EBP-n
- Restore ESP/EBP on exit
The stack

Task

• Compile a simple program (test.c)
  - Makefile (options for simpler assembly)

• Disassemble with gdb
  - GDB tutorial

• Read the assembly of `main`, `foo` (it’s simple!)
  - Understand the stack management procedure in the assembly code

• Modify test.c, observe changes in the code
Buffer overflow

- Input written to a local buffer in the stack
- Large input: continu writing outside the frame
- Overwrite the saved EBP and the return addr
- No segfault: this is our own memory
- Return: follow the overwritten address
  - this will likely segfault!
**Buffer overflow**

**Task**: observe a buffer overflow

- Read and compile target.c
  - use `-fno-stack-protector` `-zexecstack`
  see the Makefile!
- Provide large input, observe crash
- Execute **step-by-step** with gdb
  - Observe the **return address** (EBP+4) before and after the overflow
  - Observe the crash when the function returns (not during the overflow)
Shellcode

- **Goal**: execute a bash shell
  (provides easy access to all resources)

- Such a malicious code is called **shellcode**

- **Task**: write a shellcode in C
  - (We’ll write in assembly later)
  - Use `execve`
    - Optionally follow by `exit(0)` to always exit cleanly
  - Example: `shellcode.c`
**Task:** disassemble the shellcode

- Use `gdb` to disassemble `execve`, `_exit`
  - understand the system cals

TODO list for the assembly code:

1. Data needed in memory
   - string "/bin/sh"
   - The address of array with {"/bin/sh", NULL}
Task: disassemble the shellcode

2 To call execve
   - EAX <- 0xb (code of execve syscall)
   - EBX <- the address of "/bin/sh"
   - ECX <- the address of the array
   - EDX <- NULL
   - Execute call *%gs:0x10 (or int $0x80)

3 To exit
   - EAX <- 0xfc (or 0x1)
   - EBX <- 0x0 (exit code)
   - Execute call *%gs:0x10 (or int $0x80)
Problem

- We need "/bin/sh" in memory
- We can put it in the buffer
- But we don’t know its address!

Solution

- call pushes EIP in the stack
- So we can jump right before "/bin/sh" (relative jump!)
- call back
- and pop the address we need
Solution: assembly

jmp label_binsh // jmp to the call instruction at the end
label_back:
popl %esi // the address of /bin/sh is now in %esi!

...main shellcode...

label_binsh:
call label_back // jump back after pushing EIP
.string "/bin/sh" // write "/bin/bash" in the executable
Shellcode

**Task:** write the assembly shellcode

- Straightforward implementation of the TODO list
  - Using also the jump trick

- Try it yourself, or look at shellcodeasm.c

**Beware**

- The machine code should not contain 0s
- Cause most functions that overflow buffers (strncpy, etc) stop at 0s!
- So: change `movl $0x0 %eax` to `xorl %eax, %eax`, etc
Shellcode

**Task:** get the machine code

- Disassemble shellcodeasm’s `main` with gdb
- Find the `address` of the shellcode
  - the first `jmp` command
- Find the `length` of the shellcode
  - until the end of the `/bin/bash` string (without the `\0`)
- Get the machine code with gdb:
  \[ x/\text{<length>}hb \text{<address>} \]
Shellcode

**Task:** test the shellcode

- Use shellcodetest.c
- Add the shellcode in binary form
- Direct test
  - directly set a function’s return address to the buffer
- Overflow test
  - set the function’s return addresses by overflowing our own buffer
  - buffer content
    
    `<buffer-address>
    ...
    `<buffer-address>`
    `<shellcode>`
• We are almost ready!
  - We have already overflown our own buffer

• BUT
  - We had to put `<buffer-address>` in the buffer
  - We `don’t know` the buffer’s address in the target

• Solution
  - Guess it!
  - Start from ESP in a test program, add an offset
  - Try different offsets until we get lucky
Task: try this attack

- See exploit1.c
- Try different offsets until you get lucky
- Or write a script that does it
- Or cheat by having target.c print its buffer address
- Make sure to disable ASLR (see Makefile)
Can we do better?

- Goal: tolerate incorrect guesses of buffer-address

- Solution
  - Write NOPs before the shellcode
  - If execution starts there, it will reach the shellcode

```plaintext
<buffer-address>
...
<buffer-address>
<shellcode>
NOP
...
NOP
```
**Task**: try this attack

- See exploit2.c
- Try again different offsets
  - Success should be easier
Counter-measures

Canaries

• Write some value (canary) after the return value
  - CR,LF,0,-1
  - Random

• Buffer overflow still happens
  - but it overwrites the canary -> detection!

• gcc does this by default
  - Try the attack without -fno-stack-protector

• Attacks that don’t overwrite the return address still possible
Counter-measures

Non-executable stack

- Don’t allow execution of stack code
- Needs hardware/OS support
- Linux on modern processors does this by default
  - Try the attack without -zexecstack
- Return to pre-existing code in the program or a library (eg libc) still possible
Counter-measures

Address space layout randomization (ASLR)

- Randomize the stack’s address
- Makes it harder to guess `<buffer-address>`
- Linux does this by default
  - Try the attack with `echo 1 > /proc/sys/kernel/randomize_va_space`
- Needs a sufficiently large range (16-bits not enough)
References

- Aleph One, *Smashing The Stack For Fun And Profit*
- GDB tutorial : debug/disassemble C programs using gdb
- Dieter Gollmann, *Computer Security*, Section 10.4
- c0ntex, *Bypassing non-executable-stack during exploitation using return-to-libc*
- Shacham et al, *On the Effectiveness of Address-Space Randomization*