
Secure C++ Programming
Dublin, May 13–14, 2019
Scanrail1/shutterstock

Introductions

Each person please say:

- Their name
- What is their background
- How much C/C++ experience they have
- The main thing they want to learn

Me: Mihály Bárász <mihaly.barasz@nilcons.com>

Schedule

Today

- Assembly
- Basic vulnerabilities
- Some philosophy
- Most important generic protections
- Down-to-the-metal exercises

Tomorrow

- More vulnerabilities
- Best practices
- More philosophy
- More exercises

Assembly Basics

Hello World in x86/x86-64 Asm

Resources

Learning & Reference

Online references

- https://www.felixcloutier.com/x86/
- https://www.aldeid.com/wiki/X86-assembly/Instructions
  (less comprehensive, more readable)
- https://defuse.ca/online-x86-assembler.htm
  (online assembler/disassembler)
- http://www.sandpile.org/

Learning, Books

- https://en.wikibooks.org/wiki/X86_Assembly
- Programming from the Ground Up, Jonathan Bartlett
  old, 32 bit, but very good
  https://savannah.nongnu.org/projects/pgubook/

Syntax

Intel vs AT&T syntax

- Common on Linux, default syntax for most Linux tools: gcc, as, gdb...
- `mov %rax, %rbx`
  Order: source, destination
  Registers prefixed with %
- `mov $0x41, %rax`
  Constants (immediates) prefixed with $; C syntax for hexadecimals

- Universal on Windows, used in many popular Linux tools (`{n,y,f}asm`), can be switched to in most tools
- `mov rbx, rax`
  Order: destination, source
- `mov rax, 41h`
  Special syntax for hexadecimals (often `0x` can be used too)

Syntax...

- `mov $symbol, %rax`
  Loads the address of `symbol`
- `mov symbol, %rax`
  Loads the content of `symbol`

- `mov rax, symbol`
- `mov rax, [symbol]`

Memory references:

- `mov 0x18(%rdi,%rsi,4), %rax`
  displacement(base register, offset register, scalar)
- `movb $5, (%rax)`
  `movb $5, (%rax)`
  `b`, `w`, `l`, `q` suffix to the opcode

- `mov rax, [rdi + 18h + 4*rsi]`
  [base register + displacement + scalar*offset register]
- `mov qword [rax], 5`
  `mov byte [rax], 5`
  `byte`, `word`, `dword`, `qword` prefix for the operand

Syntax...

Mnemonic differences

- `cwd`, `cdq`, `cqo`, ...
- Comments prefixed with #
- Directives prefixed with `.`

- `cwtd`, `cdtl`, `cqto`, ...
- Comments prefixed with `;`
- Directives have no prefix

Syntax example

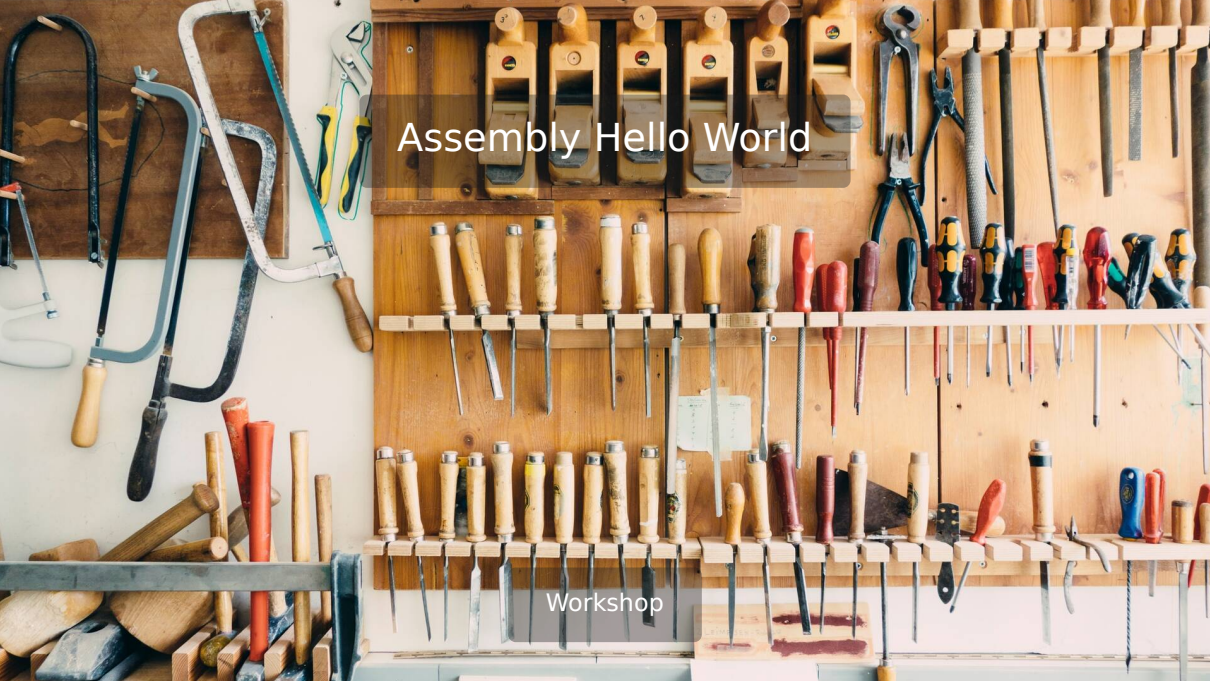
```c
int example(long long a, int *b) {
  return a / b[5];
}
```

```
.section .text

.globl example
example:
  # First argument to %rax and sign extend to %rdx
  movq    %rdi, %rax
  cqto
  # b[5] sign extended to %rcx
  movslq  20(%rsi), %rcx
  idivq   %rcx
  # Result is in %rax where we need it
  ret
```

```
section .text

global example
example:
  ; First argument to rax and sign extend to rdx
  mov    rax, rdi
  cqo
  ; b[5] sign extended to rcx
  movsxd rcx, dword [rsi+20]
  idiv   rcx
  ; Result is in rax where we need it
  ret
```


Assembly Hello World
Workshop

Assembly Workshop

Hello World!

Goal: get comfortable with the environment and assembly on Linux

Let's write following simple programs:

- Exit with some fix exit code (no crashing!)
- Exit with the exit code equal to number of command line arguments
  Hint: debug and inspect the top of the stack
- Classic "Hello World"

Additionally:

- Try both syntaxes
- Try 32 and 64 bit executables
- Learn how to get just the actual code part

Assembly Workshop…

Linux syscall ABI

64 bit

- Syscall number in `rax`
- Arguments: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`
- Instruction: `syscall`
- Result in `rax`
- Note: clobbers `rcx` and `r11`

32 bit

- Syscall number in `eax`
- Arguments: `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`
- Instruction: `int 0x80`
- Result in `eax`
- Preserves registers

Assembly Workshop...

Assembling and linking

```
as -o program.o program.s
ld -o program program.o
```

```
as --32 -o program.o program.s
ld -m elf_i386 -o program program.o
```

```
yasm -f elf64 program.s
ld -o program program.o
```

```
yasm -f elf32 program.s
ld -m elf_i386 -o program program.o
```

Extracting the code

```
objcopy -O binary -j .text program program.bin
```

```
yasm -f bin program.s -o program.bin
# BITS 32 or BITS 64 directive is required!
```

Stack Buffer Overflows

Introduction

Stack Buffer Overflows

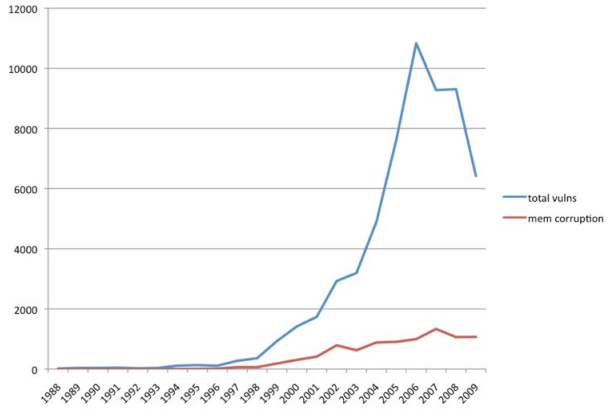
Brief history

- *1972* – US Air Force study
- *1988* – Morris worm
  CERT formed
- *1995* – “How to Write Buffer Overflows”
- *1996* – “Stack Smashing”
- *2003* – Blaster worm

“Memory Corruption Attacks The (almost) Complete History”
https://media.blackhat.com/bh-us-10/whitepapers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf

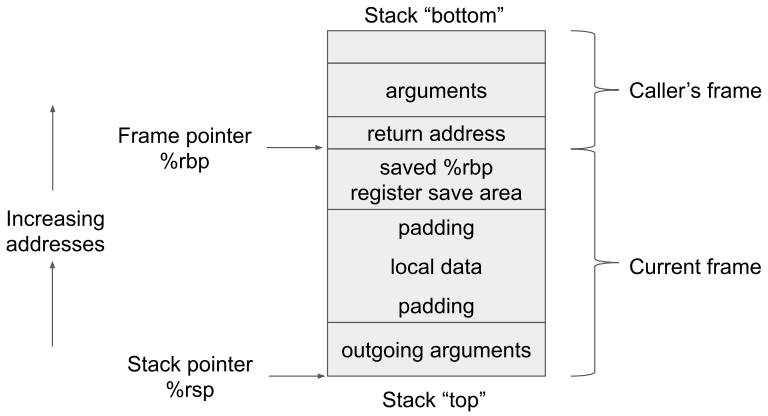
Stack Buffer Overflows

Brief history



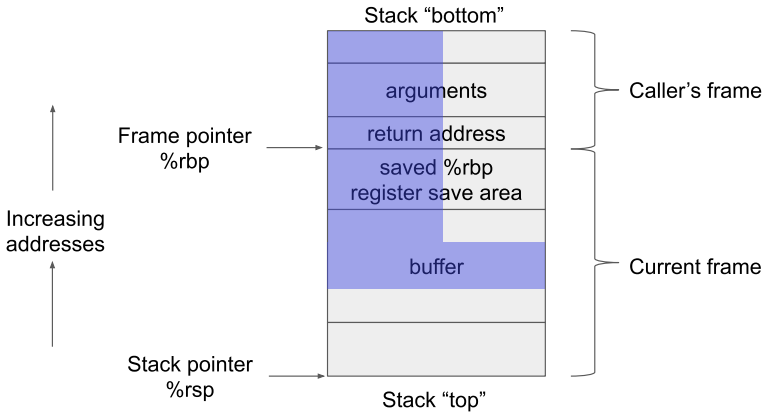
Stack Frame

Stack Frame in GCC



Stack Frame

Buffer Overflow



Stack Frame

Function Prologue and Epilogue

```
push   %rbp            Save frame pointer
movq   %rsp, %rbp      Initialize frame pointer
pushq  %rbx            Save non-volatile registers
pushq  %r12            clobbered by function
pushq  %r13
subq   $48, %rsp       Allocate stack frame

...                    Function body

addq   $48, %rsp       Deallocate stack frame
popq   %r13            Restore registers
popq   %r12
popq   %rbx
leave                  Restore %rbp
                       and deallocate stack
ret
```

Stack Frame

Function Prologue and Epilogue (Actually)

```
movq    %rbx,-24(%rsp)    Save registers
movq    %r12,-16(%rsp)
movq    %r13,-8(%rsp)
subq    $72, %rsp         Allocate stack frame

...                       Function body

movq    48(%rsp),%rbx     Restore registers
movq    56(%rsp),%r12
movq    64(%rsp),%r13
addq    $72, %rsp         Deallocate stack frame
ret
```

Stack Frame

Example

```c
#include <stdio.h>
#include <string.h>

void greet(char *name) {
  char buf[20] = "Hello: ";
  strcat(buf, name);
  puts(buf);
}
```

Stack Frame

Example

```
greet:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    $0x6c6c6548, -28(%ebp)
        movl    $0x203a6f, -24(%ebp)
        movl    $0, -20(%ebp)
        movl    $0, -16(%ebp)
        movl    $0, -12(%ebp)
        subl    $8, %esp
        pushl   8(%ebp)
        leal    -28(%ebp), %eax
        pushl   %eax
        call    strcat
        addl    $16, %esp
        subl    $12, %esp
        leal    -28(%ebp), %eax
        pushl   %eax
        call    puts
        addl    $16, %esp
        nop
        leave
        ret
```

```
gcc -m32 -O0 -S -fno-stack-protector -fno-pie
```

Stack Frame

Example

```
greet:
        pushl   %ebx
        subl    $48, %esp
        movl    $0x6c6c6548, 20(%esp)
        movl    $0x203a6f, 24(%esp)
        movl    $0, 28(%esp)
        movl    $0, 32(%esp)
        movl    $0, 36(%esp)
        pushl   56(%esp)
        leal    31(%esp), %eax
        leal    24(%esp), %ebx
        pushl   %eax
        call    strcpy
        movl    %ebx, (%esp)
        call    puts
        addl    $56, %esp
        popl    %ebx
        ret
```

```
gcc -m32 -O2 -S -fno-stack-protector -fno-pie
```

Stack Frame

Example

```
greet:
        pushl   %ebp
        xorl    %eax, %eax
        movl    $3, %ecx
        movl    %esp, %ebp
        pushl   %edi
        leal    -20(%ebp), %edi
        subl    $44, %esp
        movl    $0x6c6c6548, -28(%ebp)
        rep stosl
        leal    -28(%ebp), %edi
        pushl   8(%ebp)
        pushl   %edi
        movl    $0x203a6f, -24(%ebp)
        call    strcat
        movl    %edi, (%esp)
        call    puts
        addl    $16, %esp
        movl    -4(%ebp), %edi
        leave
        ret
```

```
gcc -m32 -Os -S -fno-stack-protector -fno-pie
```

Stack Smashing

Workshop

Stack Smashing Workshop

Goal: get first-hand experience

- All protections turned off
- Explore GCC stack organization on x86-64
- Very simple arc injection
- Basic shell code

Buffer Overflows

Mitigation

What can you do as a programmer?

- Use `std::string`
- Use modern, high abstraction C++ (e.g. `std::array`)
- Use safe variants `strncpy`, `strlcat`, ...
- Aggressively document and check bounds and invariants
- Undefined behavior — **No!**

Undefined Behavior

Examples of undefined behavior are memory accesses outside of array bounds, signed integer overflow, null pointer dereference, modification of the same scalar more than once in an expression without sequence points, access to an object through a pointer of a different type, etc. Compilers are not required to diagnose undefined behavior (although many simple situations are diagnosed), and **the compiled program is not required to do anything meaningful**.

https://en.cppreference.com/w/cpp/language/ub

Buffer Overflows

Mitigation

SEI CERT standard

https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

- Familiarize yourself to know the extent
- When in doubt read the rule(s) and weigh the options

C++ Core Guidlines

http://isocpp.github.io/CppCoreGuidelines/

And more...


Security Mindset
Engineering vs Hacking

Security Mindset

Philosophising

- Our brains are faulty
- Our models are faulty
- Our training is bad

Security Mindset

Brains

Security Mindset

Models

“Secure Perimeter”

- is almost always wrong

Code reviews / Testing

- Don’t do “security” unless you specifically train for that

Security Mindset

Brains

You don’t see your own blindspots

Security Mindset

Models

"You are not a target"

- is almost always wrong

Security Mindset

Training

Best practices

- Don’t work without the security mindset

Bruce Schneier: The Security Mindset

Security Mindset

Example

```c
#define LOG_INPUT_SIZE 80

int outputFilenameToLog(char *filename) {
  int success;

  // buffer with size appropriate for logging
  char buf[LOG_INPUT_SIZE];

  // copy filename to buffer
  strcpy(buf, filename);

  // save to log
  success = saveToLog(buf);

  return success;
}
```

Security Mindset

Example

```c
#define LOG_INPUT_SIZE 80

int outputFilenameToLog(char *filename, int length) {
  int success;

  // buffer with size appropriate for logging
  char buf[LOG_INPUT_SIZE];

  // copy filename to buffer
  strncpy(buf, filename, length);

  // save to log
  success = saveToLog(buf);

  return success;
}
```

Security Mindset

Example

```
int outputFilenameToLog(char *filename, int length) {
  int success;

  std::string buf = filename;

  // save to log
  success = saveToLog(buf.c_str());

  return success;
}
```

Security Mindset

Example

```c
#define LOG_INPUT_SIZE 80

int outputFilenameToLog(char *filename, int length) {
  int success;
  char buf[LOG_INPUT_SIZE];

  if (length > LOG_INPUT_SIZE - 1)
    return -1;

  strncpy(buf, filename, length);
  buf[length] = 0;

  success = saveToLog(buf);
  return success;
}
```

Security Mindset

Example

```c
#define LOG_INPUT_SIZE 80

int outputFilenameToLog(char *filename, int length) {
  int success;
  char buf[LOG_INPUT_SIZE];

  if (length > LOG_INPUT_SIZE - 1)
    return -1;

  strncpy(buf, filename, LOG_INPUT_SIZE - 1);
  buf[LOG_INPUT_SIZE - 1] = 0;

  success = saveToLog(buf);
  return success;
}
```

Security Mindset

Brains

Engineer vs Hacker

- Coffee machine story

Executable-space protection

W^X

History

- First in OpenBSD in 2003
- Windows XP, 2004
- Linux
  - PaX: 2000
  - Exec Shield: 2003
- NX bit in AMD’s 64 bit CPUs
- Enabled by default in all modern 64bit distros
- In 32bit mode enabled with PAE

W^X

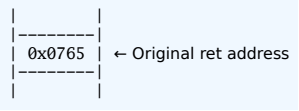
Explore

- Check binaries and .so's with `readelf -l`
- Check running processes: `/proc/<pid>/maps`
- Our "target" from Workshop 2, what if we remove the `-z execstack`
  - Would arc injection work?
  - Would shell code work?
  - How does it fail?

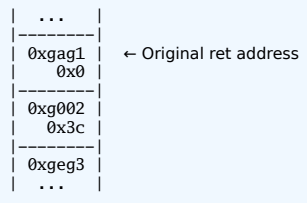
How much protection does it provide?

Say we find a stack buffer overflow. We can't run a shell code unless we un-`mprotect` it, but we can't just call `mprotect` because we can't run arbitrary code. So, we can't do too much right?

Return Oriented Programming



Return Oriented Programming




```
0xgag1: pop %rdi
        ret
```

```
0xg002: pop %rax
        ret
```

```
0xgeg3: syscall
```

Return Oriented Programming

Discussion

- All executable memory regions can be used
  - Main program's `.text`
  - Shared libraries
- Not security specific libraries rarely checked for presence of rop gadgets

Return Oriented Programming

Demo

Try the `ropper` tool

Stack Protector

Detect stack smashing

Stack Protector

History

- StackGuard: 1997
- GCC
  - 2001 – 2005 ProPolice (IBM)
  - 2005: reimplemented `-fstack-protector` and `-fstack-protector-all`
  - 2012: `-fstack-protector-strong`
  - On by default in some distros (Arch, Fedora – strong), but not in others (Debian)

Stack Protector

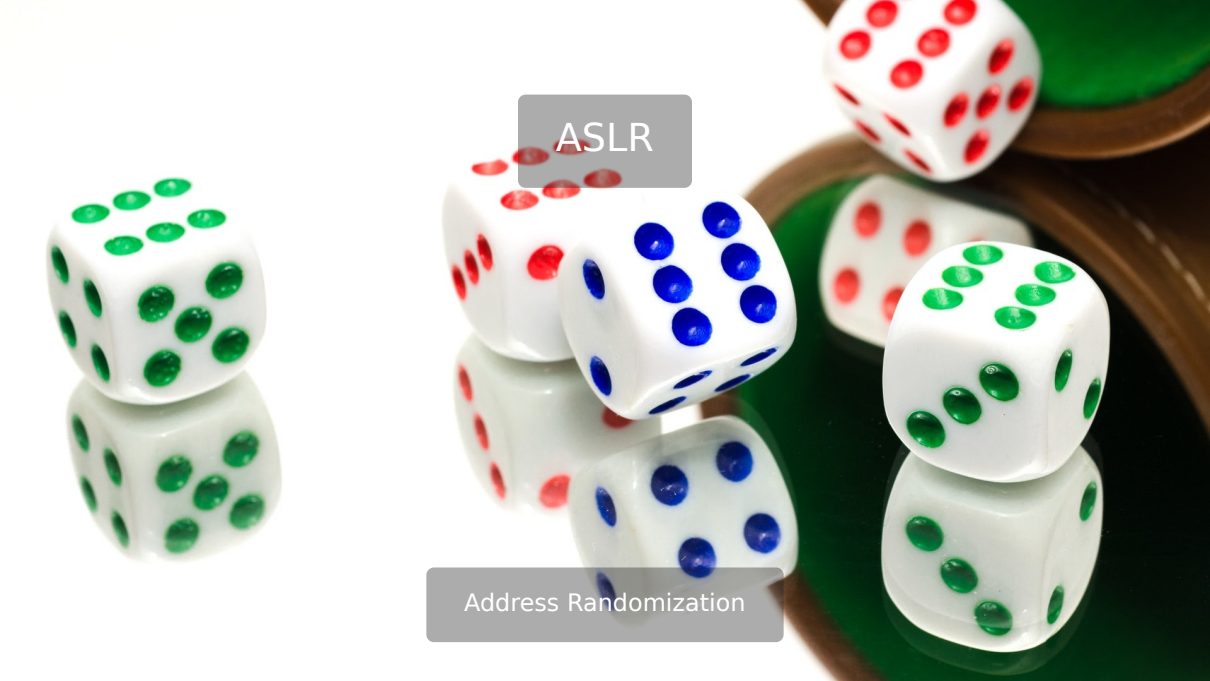
Explore

- Check Workshop 2's "target" without `-fno-stack-protector`
- Check with `-fstack-protector` and `-fstack-protector-all`
- Godbolt demo

Stack Protector

Discussion

- Only protects the return address
- Local function pointers
  - Tools to rearrange local variables
- Initialized once for the lifetime of a thread


ASLR
Address Randomization

Address Space Layout Randomization

History

- Linux
  - 2001: PaX project
  - 2002: full implementation
  - 2005: enabled by default (2.6.12)
  - ????: PIE by default
- OpenBSD: 2003 first ASLR by default, 2008 first to PIE by default
- Windows: 2007, Vista (not for old binaries)
- Mac OS X: 2007 system libs, 2011 for all apps
- iOS and Android: 2011

ASLR

What and how

- What is randomized?
- What isn’t randomized?
- Randomized by how much?

ASLR

Explore

Goal: learn details of ASLR in Linux

- Build (or choose) a PIE executable
- Observe its memory map over multiple runs
- How many independently randomized segments are there?
- What are they?
- Write your own C or C++ program and check your findings “from within”
- Repeat for 64 and 32 bits

Additionally

- What about non-PIE executables?

ASLR

How to defeat

- Reduce entropy
  - Heap spraying and NOP slides
- Information leak
  - Through other vulnerability
  - Side channel
    - Jump Over ASLR
    - ASLR ⊕ Cache
- Brute force
  - Mitigation: throttle crashes

ASRL

Linux details

Performance

- On x86-32 there is quite a price (read)
- x86-64: much better, hardware support for PC-relative addressing, but not free

Statically linked binaries

- Limited ASLR
- Build a dynamic binary which has everything statically linked

ASRL

Enable/disable

```
echo 0 > /proc/sys/kernel/randomize_va_space
sysctl -w kernel.randomize_va_space=0
```

- 0: no randomization
- 1: conservative randomization (shared libs, stack, mmap, VDSO)
- 2: full randomization (brk() aka. heap too)

`personality` syscall to change execution domain for a single process (and its children)

```
setarch `uname -m` -R /bin/bash
```

GDB turns it off by default for easier debugging! Turn back on with `set disable-randomization off` (or simply `aslr on` with GEF and co.)


Security is a Trade-off

All or nothing?

Security is not all or nothing

Have to prioritize it together with

- costs
- speed of development
- maintainability
- performance
- code quality
- …

Performance

- Many of the general protections are cheap, but not zero cost
- Example: using `std::string` for all processing results in many small strings on heap; might result in significant performace hit
- Redundant validation

Solution: profile don’t guess!

Development time & costs

- Cognitive load can be significant
- Solution
  - Traning. Learn the patterns, become more confident and productive, less load
  - Break things up. Not all phases are equally important
    - More focus during design
    - Less during development
    - More during review and testing
- Pays off with code quality and maintainability
  - Biggest enemy: complexity

Flexibility & Usability

- Requires a lot of self-discipline
- The UX design have to be involved and security minded
- Risk modelling and mitigation

Heap Buffer Overflows

and other vulnerabilities

Heap vulnerabilities

Overview

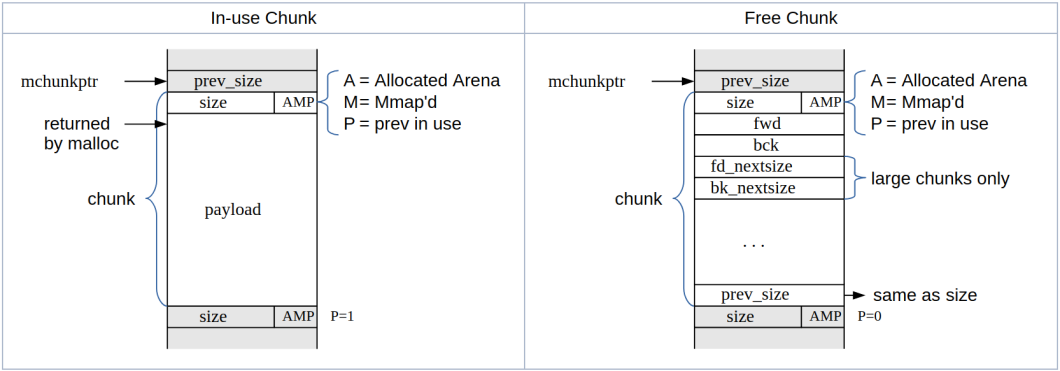
Buffer Over/underflows

- Overwrites other data on the heap
- Overwrites memory managment internal bookkeeping data

Improper handling

- Use after free
- Double `free` / `delete`
- Mixing `malloc`/`new` and `free`/`delete`
- Mixing `new`/`new[]` and `delete`/`delete[]`
- ...
- Corrupts memory managment internal bookkeeping data

Malloc internals



https://sourceware.org/glibc/wiki/MallocInternals

Malloc internals

unlink

```
#define unlink(P, BCK, FWD) { \
  FWD = P->fwd; \
  BCK = P->bck; \
  FWD->bck = BCK; \
  BCK->fwd = FWD; \
}
```

Heap vulnerabilities

Mitigation

- General buffer overflow prevention
- Smart pointers

Tools:

- Valgrind
- GCC & Clang both have superb built-in sanitizers
  - `-fsanitize=address -fno-omit-frame-pointer`
  - https://github.com/google/sanitizers/wiki/AddressSanitizerFlags

Heap vulnerabilities

Explore

- Does `operator new` calls malloc? How?
- What does this code compile to? What are the different parts?

```
int *allocate(int num) {
  return new int[num]();
}
```

- Write a program that allocates a few chunks, fills them with something easily recognizable, then debug it and investigate how it all looks...


Format String Vulnerabilities
printf(☠)

Format String Vulnerabilities

History

- Exploded in 2000
- Considered “bug”, easy to detect
- Full remote root in many cases

Format String Vulnerabilities

What's the problem with this code:

```
char tmpbuf[512];

snprintf (tmpbuf, sizeof (tmpbuf), "login attempt: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = '\0';
syslog (LOG_NOTICE, tmpbuf);
```

Format String Vulnerabilities

What's the problem with this code:

```
char tmpbuf[512];

snprintf (tmpbuf, sizeof (tmpbuf), "login attempt: %s", user);
tmpbuf[sizeof (tmpbuf) - 1] = '\0';
fprintf (log_file, tmpbuf);
```

Format String Vulnerabilities

Basically, this is always wrong and a vulnerability:

```
printf_like_function(attacker_controlled_string);
```

Variadic Functions

```c
#include <stdio.h>
#include <stdarg.h>

int FindMax(int n, ...) {
  int i, val, largest;
  va_list vl;
  va_start(vl, n);
  largest = va_arg(vl, int);
  for (i = 1; i < n; i++) {
    val = va_arg(vl, int);
    largest = (largest > val) ? largest : val;
  }
  va_end(vl);
  return largest;
}

int main() {
  int m;
  m = FindMax(7,702,422,631,834,892,104,772);
  printf("The largest value is: %d\n",m);
  return 0;
}
```

Format String Vulnerabilities

How to exploit

- Look arbitrarily up the stack: `"%p%p%p%p%p%p%p%p"`
- Crash (denial of service): `"%s%s%s%s%s%s%s%s"`
- Write arbitrary* memory: `%n`

*: with many limitations

- Format string *or* buffer we are writing to have to be on the stack (and not too far)
- Address can't contain 0 bytes. Makes it significantly harder (if not impossible) on x86-64

Format String Vulnerabilities

Explore

- Write a small program with which you can explore bad format strings
- Walk up the stack, can you identify all the pieces?
- Make sure you have stack canaries…
  - What can you tell about stack canary values? Why might that be?

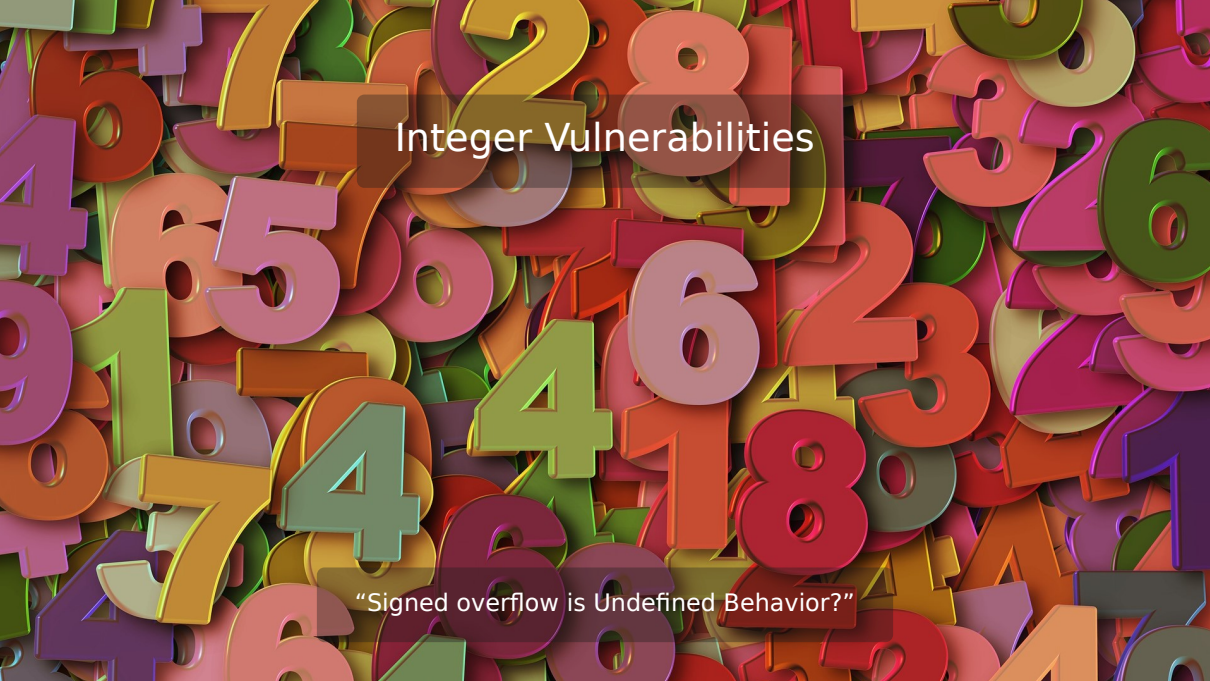
Format String Vulnerabilities

Mitigation

- Format strings should be constant literals
  - OK, *maybe* allow `_("format")` with `gettext`
- Static analysis tools detect these well

Generally for variadic functions

- Don’t use variadic functions
- Use variadic templates



Integer Vulnerabilities

“Signed overflow is Undefined Behavior?”

Integer Vulnerabilities

- Integer conversion/promotion rules are *complex*

  “Well, at the time we thought compatibility with B was important, so we kept that; I wish we hadn’t, no one cares about B anymore.”

- Many things are “Implementation Defined”
  - Can cause problems much later than the code was written
- Hard to reason about
  - `-INT_MIN` is `INT_MIN`?

Integer Vulnerabilities

Examples

```
void getComment(size_t len, char *src) {
  size_t size;
  size = len - 2;
  char *comment = (char *)malloc(size + 1);
  memcpy(comment, src, size);
  return;
}
```

Integer Vulnerabilities

Examples

```
p = calloc(sizeof(element_t), count);
```

Integer Vulnerabilities

Examples

```
if (off > len - sizeof(type_name)) goto error;
```

Integer Vulnerabilities

Examples

```
void initialize_array(int size) {
  if (size < MAX_ARRAY_SIZE) {
    array = malloc(size);
    /* initialize array */
  } else {
    /* handle error */
  }
}
```

Integer Vulnerabilities

Mitigation

- Check the ranges obsessively
- Use `size_t` for sizes of objects
  - Or at least `unsigned` types
  - C11 Annex K: `rsize_t`
- User defined types with better semantics
  - User defined literals!
  - `5_hour + 10_kg`

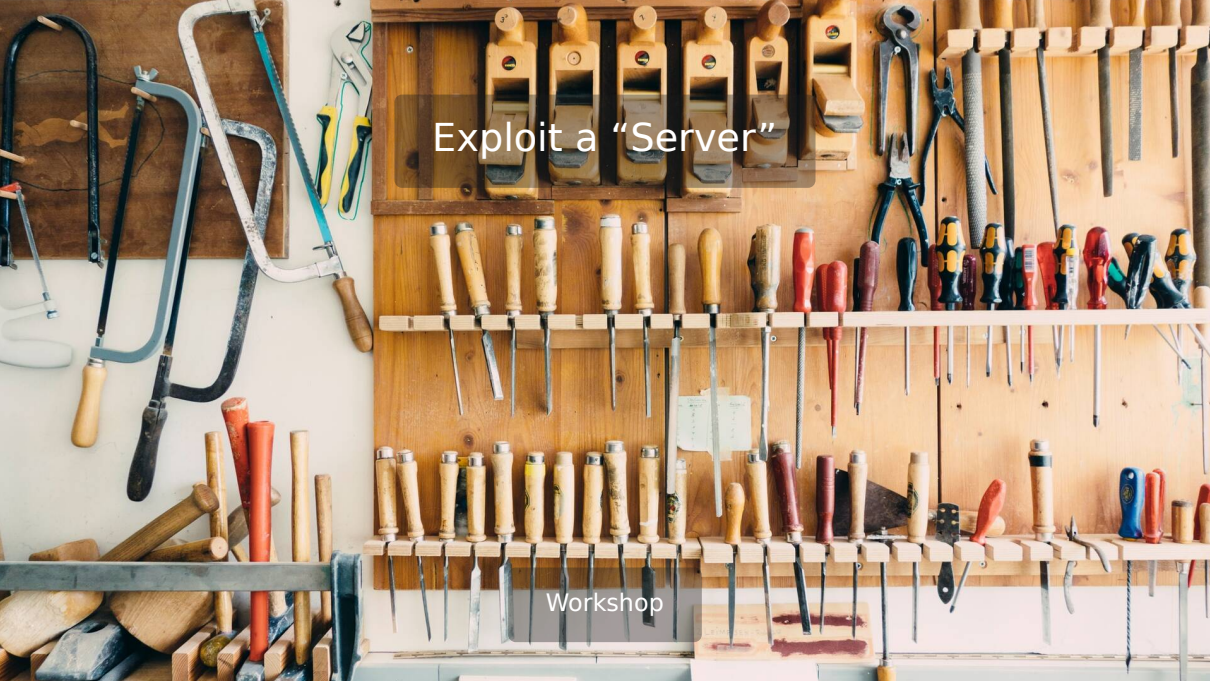

FORTIFY_SOURCE
Fixing your buffer overflows for you


RELRO
RELocation Read-Only


Control-Flow Enforcement Technology

Control-Flow Enforcement Technology

- Intel Spec
- https://lwn.net/Articles/758245/


Exploit a “Server”
Workshop

“Server” exploit

Goal: to see a (reasonably) real-world example with all of the protections enabled, which can nevertheless be exploited.

Best Practices

What to do

Best Practices

> Within C++, there is a much Small and cleaner language struggling to get out.
>
> Bjarne Stroustrup

Best Practices

Within C++, there is a much Small and cleaner language struggling to get out.

Bjarne Stroustrup

And no, that Small and cleaner language is not Java or C#.

Best Practices

SEI CERT standard

https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

- Familiarize yourself to know the extent
- When in doubt read the rule(s) and weigh the options

C++ Core Guidlines

http://isocpp.github.io/CppCoreGuidelines/

And more...

Best Practices

Run tests with many different setups

- Different compilers
- Different optimization options
- Memory constrained (?)

Best Practices

Not allocate with new

- Smart pointers

Best Practices

Not allocate with new

- Smart pointers

Shared pointers stink

- If they start to spread — time to think

Best Practices

- Prefer `enum class` to `enum`s
  - No undefined behavior of converting `int` to enum out of range

Best Practices

- String literals: `"foobar"s`
- User defined literals: `5_hour + 10_kg`

Best Practices

- Check the ranges obsessively
- Use `size_t` for sizes of objects
  - Or at least `unsigned` types
  - C11 Annex K: `rsize_t`

C11, Annex K

Secure variants of many functions

- `memcpy_s`, `strcat_s`, ...
- `gets_s`, `printf_s`, ...

- Not in glibc
- Generally poor adoption
  - Considered for removal
  - Interface & ease of adoption matters
  - Field Experience With Annex K — Bounds Checking Interfaces

C11, Annex K

Continued

`memset` is sometimes used to zero-out sensitive memory

- Often optimized out if the memory “cannot” be accessed later
- `memset_s` is specified to *not be* optimized out
- Research alternatives on your platform if you need this

Compilers are Evil

Or maybe specs are

```
void GetData(char *MFAddr) {
  char pwd[64];
  if (GetPasswordFromUser(pwd, sizeof(pwd))) {
    ...
  }
  memset(pwd, 0, sizeof(pwd));
}
```

Undefined Behavior is Evil

Or maybe compilers are

```
#include <iostream>
#include <complex>
using namespace std;

int main(void) {
  complex<int> delta;
  complex<int> mc[4] = {0};

  for (int di = 0; di < 4; di++, delta = mc[di]) {
    cout << di << endl;
  }
}
```

Best Practices

Warnings are errors

Best Practices

Be selective with the libraries you use

- Open source libraries have more eyes on them, but also easier targets

Dynamic Analysis

GCC and Clang both have good sanitizers

- asan
- tsan
- ubsan

Espesically effective with stress and scalability tests

Best Practices

- Do stress testing

Fuzz Testing

Fuzz testing can be very effective in uncovering bugs which might lead to vulnerabilities

- Requires a significant investment (similar to TDD)

Miscellaneous

Topics we didn’t touch

Concurrency

- A world in its own
- Minimize shared data
- Thread sanitizer

Unicode

Complexity in strings

- Unicode Standard is more than 1000 pages
- Many things that seem easy are actually quite complex
  - Are two strings “equal”?
  - Different strings can still look exactly the same
  - Length of a string?

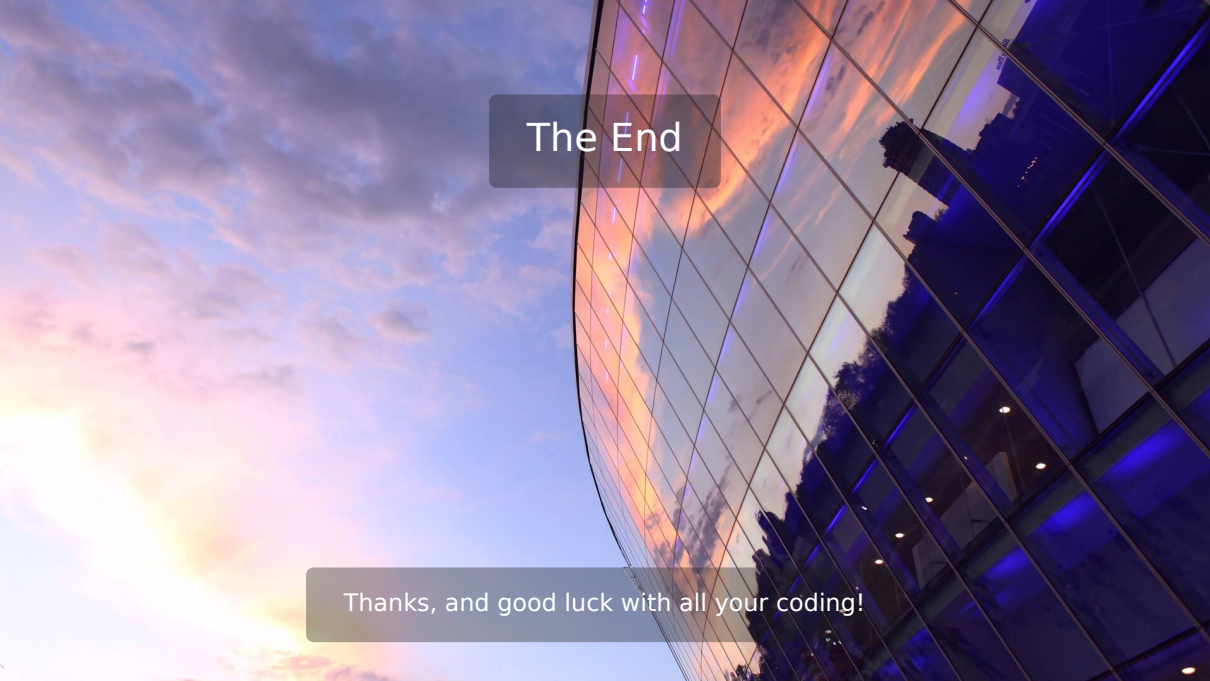
I18N in general is tricky and common source of bugs & vulnerabilities

Sensitive Information

- Logging is not neutral
  - Memory dumps…
- Lifetime management
- What gets on a disk

Secure communication

- DDoS, Men-in-the-middle, Spoofing, Byzantine failures, Side channels, oh my!
- Cryptography in general...


The End
Thanks, and good luck with all your coding!