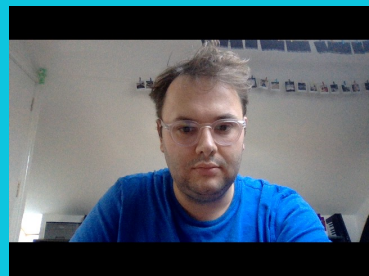
University of BRISTOL

A quick refresher on Memory Corruption and ABIs

Joseph Hallett



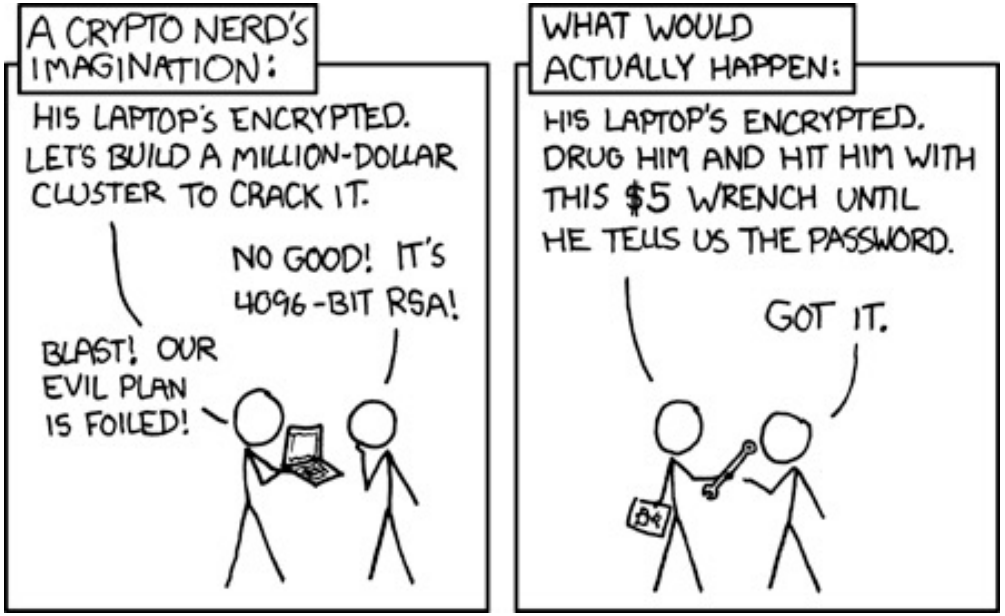
Agenda

- Motivation (non-technical)
- Background
- C to assembly (calling conventions)
- Memory layout
- Pointers to useful tools

Motivation

“Security is a chain; it’s only secure as the weakest link.” — Bruce Schneier

Motivation (xkcd)



Motivation

Cryptography and security protocols are theoretically secure…

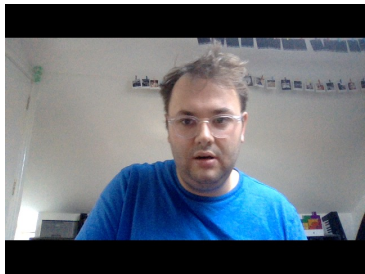
But the systems we implement them on make tradeoffs (in hardware and software; but we'll focus mostly on software)
And building computers is *really hard*

Mistaken assumptions create opportunities…

What is a program?

Something that does what it's meant to do
- Functional (intended) behaviour
- Security policy (what its not meant to do)

Unintended behaviours can include:
- Design flaws
- Bugs
- Lower-level bugs…
- Mistaken assumptions



Weaknesses and Vulnerabilities

Weakness: when a program has a flaw that allows an attacker to do something the programmer didn't anticipate, or which could cause problems later
…see MITRE's CWEs

Vulnerability: when these weaknesses can be *exploited* by an attacker to violate part of the program's design, and do something harmful

…see MITRE's CVEs

e.g. break confidentiality, integrity or availability principles

Proof of concept (exploit) usually required!

bristol.ac.uk

Weakness is not a Vulnerability

These terms are constantly conflated!

(and it's my pet peeve)

Common Weakness Enumeration

A Community-Developed List of Software & Hardware Weakness Types

Home > CWE List > CWE- Individual Dictionary Definition (4.5) ID Lookup: Go

Home | About | CWE List | Scoring | Mapping Guidance | C

News | Search

CWE-252: Unchecked Return Value

Weakness ID: 252
Abstraction: Base
Structure: Simple

Status: Draft

Presentation Filter: Complete

Description

The software does not check the return value from a method or function, which can prevent it from detecting unexpected states and conditions.

Extended Description

Two common programmer assumptions are "this function call can never fail" and "it doesn't matter if this function call fails". If an attacker can force the function to fail or otherwise return a value that is not expected, then the logic could lead to a vulnerability, because the software is not programmer assumes. For example, if the program calls a fun privileges but does not check the return code to ensure that p successfully dropped, then the program will continue to opera privileges.

bristol.ac.uk

If I don't check the return value of a call to `printf()` does that program have a vulnerability?

(no.)

Common Weakness Enumeration

A Community-Developed List of Software & Hardware Weakness Types

Home > CWE List > CWE- Individual Dictionary Definition (4.5) ID Lookup: Go

Home | About | CWE List | Scoring | Mapping Guidance | C

News | Search

CWE-252: Unchecked Return Value

Weakness ID: 252 **Status:** Draft
Abstraction: Base
Structure: Simple

Presentation Filter: Complete

Description

The software does not check the return value from a method or function, which can prevent it from detecting unexpected states and conditions.

Extended Description

Two common programmer assumptions are "this function call can never fail" and "it doesn't matter if this function call fails". If an attacker can force the function to fail or otherwise return a value that is not expected, then the logic could lead to a vulnerability, because the software is not programmer assumes. For example, if the program calls a fun privileges but does not check the return code to ensure that p successfully dropped, then the program will continue to opera privileges.

Exploits

- A program or technique that takes advantage of a vulnerability to violate the security policy

- Could be published to prove existence of a vulnerability...
- Could be used as part of malware (computer virus...)

- Proper management is vital to prevent the latter! (Responsible disclosure)

Typical Vulnerabilities

WYSINWYX: What you see is not what you execute (G. Balakrishnan et al.)

High-level code gets translated into a low-level representation

Separate variables become continuous memory addresses

Data types become bit-patterns

Memory corruption becomes a big problem

It all ends up being a big Von Neumann machine program tape!

Typical Vulnerabilities

- Over/underflow
- Data corruption
- Control flow corruption
- Denial of service

- Typically, will cause the program to crash...
- …occasionally will become an exploit

Mitigations

Put in place mechanisms that remedy the weakness, or prevent the exploitation of the vulnerability

For example, stack canaries let us spot when a stack buffer has been overflown...

...doesn't fix the buffer overflow

...but makes it *significantly more difficult to exploit*

ASLR, Shadow stacks, Sandboxing…

C

Popular programming language

Far from dead!

Everything (nearly) is built on top of it (or at least libc)

Designed for systems programming

By design unsafe
…it's the programmer's job to make sure their program is correct

Allows the programmer access to raw(ish) memory addresses (pointe

Why don't people like C?

It will always assume the programmer knows best

Limited support for anything more than primitive types

Limited support for primitive types (unsigned vs signed, int vs long)

Limited bounds checking

`Const` doesn't mean what you think it does

Lots of dangerous vulnerabilities have been found in C prograr

bristol.ac.uk

Why don't people like C?

Lots of legacy code still written in C

Some effort to rewrite in safer languages (e.g. Rust)

…This isn't necessarily a great idea.

C is very stable, very portable, and sometimes you really do need to do crazy memory tricks. Rewriting introduces new bugs and new gotchas.

Not all bugs relate to Cs unsafeness (program logic is hard)

```c
#include <stdio.h>
int get_cookie() {
        return rand(); //random number
}
int main(void) {
        int guess;
        char name[20];
        guess=get_cookie();
        printf("Enter your name:\n");
        gets(name);
        if(guess == 0x41424344)
                printf("Hurray... you win Dear %s\n",name);
        else
                printf("Sorry, Better luck next time :( \n");
        return 0;
}
```


```c
#include <stdio.h>
int get_cookie() {
        return rand(); //random number
}
int main(void) {
        int guess;
        char name[20];
        guess=get_cookie();
        printf("Enter your name:\n");
        gets(name);
        if(guess == 0x41424344)
                printf("Hurray... you win Dear %s\n",name);
        else
                printf("Sorry, Better luck next time :( \n");
        return 0;
}
```

```
uob@work-mac Desktop % ./test
Enter your name:
warning: this program uses gets(), which is unsafe.
AAAAAAAAAAAAAAAAAAAAAAADCBA
Hurray... you win Dear AAAAAAAAAAAAAAAAAAAAAAADCBA
```


Memory Layout

From low to high...

- .text (Program code)
- .plt .plt.got (Library code)
- .data (initialised data)
- .bss (uninitialised data)
- The heap (growing up)…

From high to low...

- Arguments and environment
- The stack (growing down)…

This will vary by system and OS and architecture.

Usually virtual memory aligned to page boundaries.

x86 Assembly (32-bit)

- 6 32-bit general purpose registers `eax, ebx, ecx, edx, esi, edi`
- 2 special 32-bit registers `esp, ebp`
- 1 instruction pointer (program counter) `eip`
- Sometimes more registers depending on the chip (e.g. floating point, vector, consult your CPUs programmer's manual)

- Tonnes of instructions—its a big CISC
- (that gets translated into a RISC microcode internally)
- (…unless you have one of the chips that doesn't)

amd64 Assembly (64-bit)

16 64-bit general purpose registers `rax, rbx, rcx, rdx, rsi, rdi`
`r8, r9, r10, r11, r12, r13, r14, r15`
2 special 32-bit registers `rsp, rbp`
1 instruction pointer (program counter) `rip`
Sometimes more registers depending on the chip (e.g. floating point, vector, consult your CPUs programmer's manual)

Tonnes of instructions—its a big CISC
(that gets translated into a RISC microcode internally)
(…unless you have one of the chips that doesn't)

x86/64 Assembly

- If in doubt look it up

Intel® 64 and IA-32 Architectures Software Developer's Manual

Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of three volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384. Refer to all three volumes when evaluating your design needs.

x86/64 Assembly

Lots of different assemblers for x86... each with their own syntax!

Add 10 to rax in Intel:

```
add rax,10
```

Add 10 to rax in AT&T/GNU:

```
addq $10,%rax
```

People have strong opinions about which is better...

GNU/Linux tools / Academics generally use AT&T syntax

Everyone else (and other architectures) use Intel

`set disassembly-flavor intel` (if it bothers you…)

Calling conventions

- How do you call a function from C?
- How does that translate into registers?
- How do shared libraries know where arguments go?

- Calling conventions!

- Defined by the OS, but not strictly enforced!
- (But things will break unless you know what you're doing)
- Most programming languages follow C so they can use libc...

bristol.ac.uk

x86 Calling conventions

Lots of different ones (the instruction set has been around since the 70s!)

You essentially have to look up whatever your system uses

(In the case of Windows this may be more than one!)

You need to know `cdecl`, `stdcall`, `fastcall`, `thiscall`

x86 Calling conventions

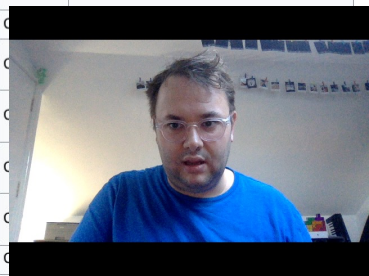
cdecl: everything goes on the stack
caller cleans up

stdcall: everything goes on the stack
callee cleans up

fastcall: pass things in registers,
`eax, edx, ecx`
then on the stack
thiscall: class pointer in `ecx` then stack

bristol.ac.uk

Archi-tecture	Name	Operating system, compiler	Parameters		Stack cleanup	Notes
			Registers	Stack order		
8086	cdecl			RTL (C)	Caller	
	Pascal			LTR (Pascal)	Callee	
	fastcall (non-member)	Microsoft	AX, DX, BX	LTR (Pascal)	Callee	Return pointer in BX.
	fastcall (member function)	Microsoft	AX, DX	LTR (Pascal)	Callee	`this` on stack low address. Return pointer in AX.
	fastcall	Turbo C[26]	AX, DX, BX	LTR (Pascal)	Callee	`this` on stack low address. Return pointer on stack high address.
		Watcom	AX, DX, BX, CX	RTL (C)	Callee	Return pointer in SI.
IA-32	cdecl	Unix-like (GCC)		RTL (C)	Caller	When returning struct/class, the calling code allocates space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address. Stack aligned on 16-byte boundary due to a bug.
	cdecl	Microsoft		RTL (C)	Caller	When returning struct/class, • Plain old data (POD) return values 32 bits or smaller are in the EAX register • POD return values 33–64 bits in size are returned via the EAX:EDX registers. • Non-POD return values or values larger than 64-bits, the calling code will allocate space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address. Stack aligned on 4-byte boundary.
	stdcall	Microsoft		RTL (C)	C	
	fastcall	Microsoft	ECX, EDX	RTL (C)	C	
	register	Delphi and Free Pascal	EAX, EDX, ECX	LTR (Pascal)	C	
	thiscall	Windows (Microsoft Visual C++)	ECX	RTL (C)	C	
	vectorcall	Windows (Microsoft Visual C++)	ECX, EDX, [XY]MM0–5	RTL (C)	C	
		Watcom compiler	EAX, EDX, EBX, ECX	RTL (C)	C	



amd64 Calling conventions

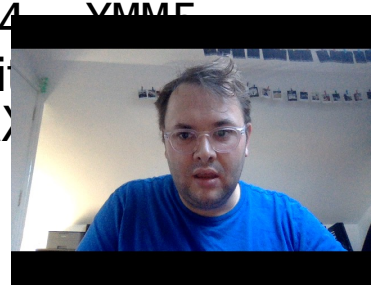
With amd64 the instruction set designers tried to sort this mess and start again
Now we only have two (well three…) calling conventions, both similar to fastcall

Microsoft x64

Registers `RCX, RDX, R8, R9` for the first four integer or pointer arguments (in that order); `XMM0, XMM1, XMM2, XMM3` are used for floating point arguments. Additional arguments are pushed onto the stack (right to left)

System V AMD64 ABI

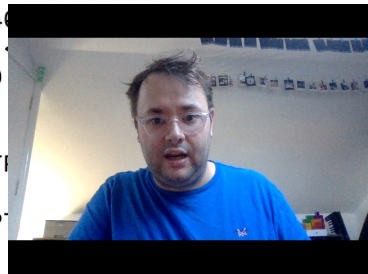
The first six integer or pointer arguments are passed in registers `RDI, RSI, RDX, RCX, R8, R9; XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6` and XMM7 are used for certain floating point arguments. Addi
arguments are passed on the stack. The return value is stored in RA



```c
#include <stdio.h>
int get_cookie() {
        return rand(); //random number
}
int main(void) {
        int guess;
        char name[20];
        guess=get_cookie();
        printf("Enter your name:\n");
        gets(name);
        if(guess == 0x41424344)
                printf("Hurray... you win Dear %s\n",name);
        else
                printf("Sorry, Better luck next time :( \n");
        return 0;
}
```

```
0000000000401156 <get_cookie>:
  401156:       55                      push   rbp
  401157:       48 89 e5                mov    rbp,rsp
  40115a:       e8 01 ff ff ff          call   401060 <rand@plt>
  40115f:       5d                      pop    rbp
  401160:       c3                      ret

0000000000401161 <main>:
  401161:       55                      push   rbp
  401162:       48 89 e5                mov    rbp,rsp
  401165:       48 83 ec 20             sub    rsp,0x20
  401169:       b8 00 00 00 00          mov    eax,0x0
  40116e:       e8 e3 ff ff ff          call   401156 <get_cookie>
  401173:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  401176:       bf 10 20 40 00          mov    edi,0x402010
  40117b:       e8 b0 fe ff ff          call   401030 <puts@plt>
  401180:       48 8d 45 e0             lea    rax,[rbp-0x20]
  401184:       48 89 c7                mov    rdi,rax
  401187:       b8 00 00 00 00          mov    eax,0x0
  40118c:       e8 bf fe ff ff          call   401050 <gets@plt>
  401191:       81 7d fc 44 43 42 41    cmp    DWORD PTR [rbp-0x4],0x41424344
  401198:       75 18                   jne    4011b2 <main+0x51>
  40119a:       48 8d 45 e0             lea    rax,[rbp-0x20]
  40119e:       48 89 c6                mov    rsi,rax
  4011a1:       bf 21 20 40 00          mov    edi,0x402021
  4011a6:       b8 00 00 00 00          mov    eax,0x0
  4011ab:       e8 90 fe ff ff          call   401040 <printf@plt>
  4011b0:       eb 0a                   jmp    4011bc <main+0x5b>
  4011b2:       bf 40 20 40 00          mov    edi,0x4
  4011b7:       e8 74 fe ff ff          call   401030
  4011bc:       b8 00 00 00 00          mov    eax,0x0
  4011c1:       c9                      leave
  4011c2:       c3                      ret
  4011c3:       66 2e 0f 1f 84 00 00    nop    WORD PT
  4011ca:       00 00 00
  4011cd:       0f 1f 00                nop    DWORD P
```



bristol.ac.uk

Useful tools

Debuggers: *GDB, LLDB*

Disassemblers: *Ghidra, Radare2, Objdump (binutils)*

Languages: *Python (other languages also have bindings)*

Hex Editors: *Radare2, XXD, emacs/vi*

Other tools for Windows (I hear OllyDbg is nice…)

Compilation Options

For GCC

- `-fno-stack-protector`
  (no stack protector)
- `-z execstack`
  (let me run shellcode off the stack)
- `-mno-accumulate-outgoing-args -mpush-args`
  (don't optimise calling conventions please)

Finding a vulnerability is the first step…

- Then do you patch it?

- Or work out how to exploit it?