SysSoft Security

A quick refresher on Memory corruption and ABI

sanjay.rawat@Bristol.ac.uk

Lecture(s) Agenda

- Motivation (non-technical)
- Background
  - C to assembly
  - Memory layout

Lecture(s) Agenda

- Motivation (non-technical)
- Background
  - C to assembly
  - Memory layout
- Example – Stack buffer Overflow

Motivation

- Bruce Schneier wrote

  "Security is a chain; it's only as secure as the weakest link."

Motivation

- Bruce Schneier wrote

  "Security is a chain; it's only as secure as the weakest link."

- cryptography is already strong (mathematically!)

Motivation

- Bruce Schneier wrote

  "Security is a chain; it's only as secure as the weakest link."

- cryptography is already strong (mathematically!)
- problem lies in software, hardware, networks etc.

Motivation

- Bruce Schneier wrote

  "Security is a chain; it's only as secure as the weakest link."
- cryptography is already strong (mathematically!)
- problem lies in software, hardware, networks etc.
- For now, we’ll focus on software parts

Motivation

- Bruce Schneier wrote

  "Security is a chain; it's only as secure as the weakest link."
- cryptography is already strong (mathematically!)
- problem lies in software, hardware, networks etc.
- For now, we'll focus on software parts

What is a computer program

- Intended behavior
  - Functional
  - Security policy/objectives

What is a computer program

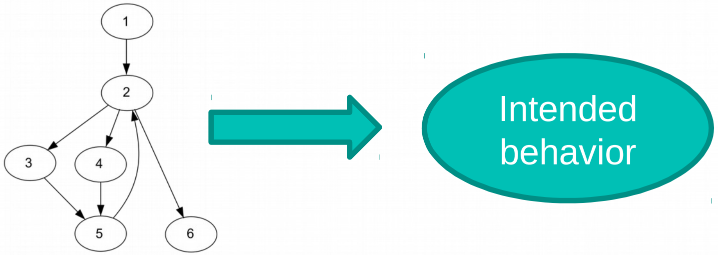
- Intended behavior
  - Functional
  - Security policy/objectives

Security feature
(crypto)

What is a computer program

- Intended behavior
  - Functional
  - Security policy/objectives

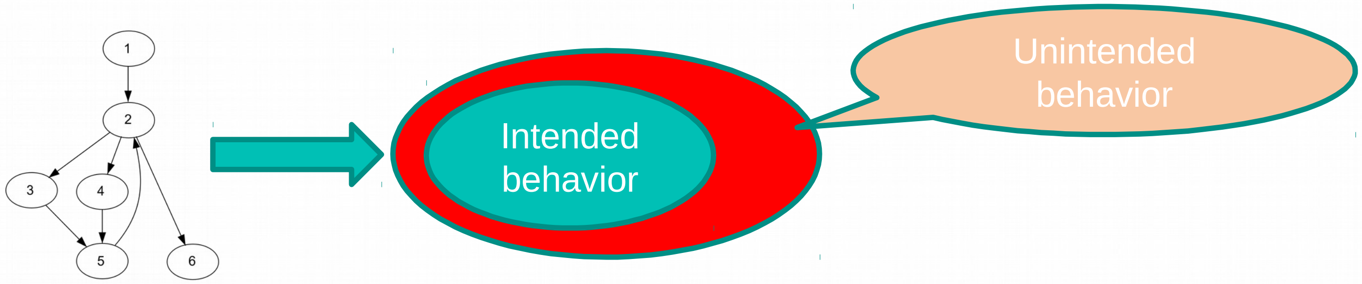
Security feature (crypto)



What is a computer program

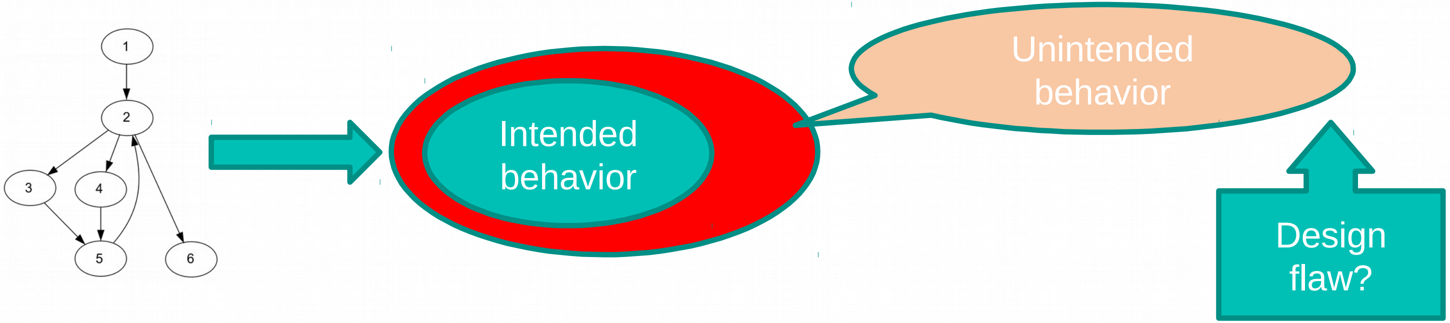
- Intended behavior
  - Functional
  - Security policy/objectives

Security feature (crypto)



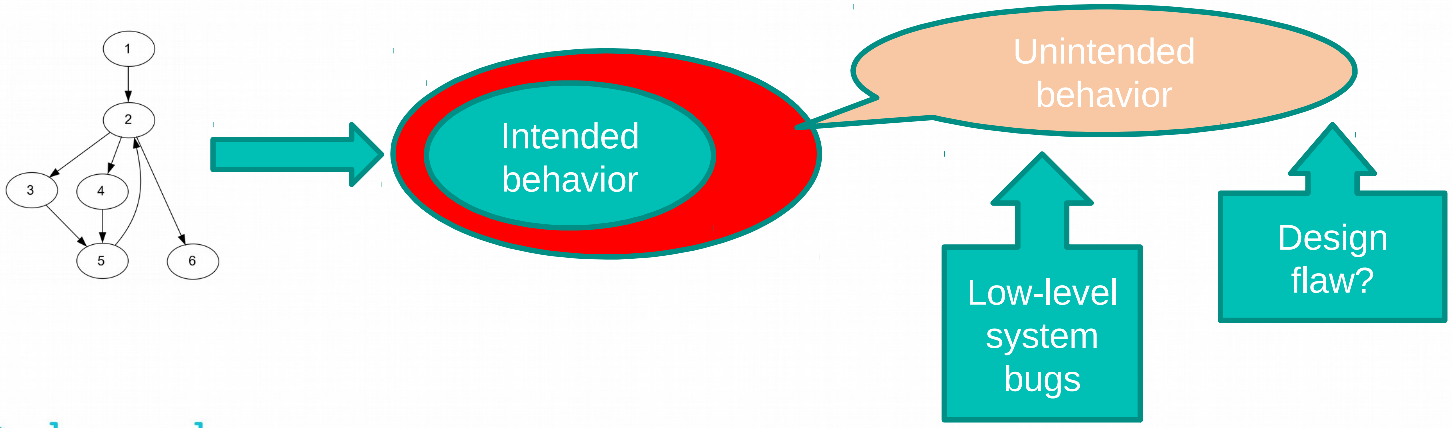
What is a computer program

- Intended behavior
  - Functional
  - Security policy/objectives



What is a computer program

- Intended behavior
  - Functional
  - Security policy/objectives



Software Vulnerability

Software Vulnerability

- WEAKNESS: a type of behavior that has the potential for allowing an attacker to violate the intended security policy, if the behavior is made accessible to the attacker. e.g. overflow via strcpy()

Software Vulnerability

- WEAKNESS: a type of behavior that has the potential for allowing an attacker to violate the intended security policy, if the behavior is made accessible to the attacker. e.g. overflow via strcpy()

- VULNERABILITY: a set of one or more related weaknesses within a specific software product or protocol that allows an actor to access resources or behaviors that are outside of that actor's control sphere, i.e., that do not provide appropriate protection mechanisms to enforce the control sphere.

Software Vulnerability

- WEAKNESS: a type of behavior that has the potential for allowing an attacker to violate the intended security policy, if the behavior is made accessible to the attacker. e.g. overflow via strcpy()

- VULNERABILITY: a set of one or more related weaknesses within a specific software product or protocol that allows an actor to access resources or behaviors that are outside of that actor's control sphere, i.e., that do not provide appropriate protection mechanisms to enforce the control sphere.

- **Vulnerability** is the intersection of three elements: a system susceptibility or flaw, attacker access to the flaw, and attacker capability to exploit the flaw (there are mitigations!)

Exploits

- Exploit:
  - An *exploit* is a piece of software or technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy.
  - Vulnerabilities in software are subject to exploitation.
  - Exploits can take many forms, including worms, viruses, and trojans.

Exploits

- Exploit:
  - Proof-of-concept exploits are developed to prove the existence of a vulnerability.
  - Proof-of-concept exploits are beneficial when properly managed (for example?).
  - Proof-of-concept exploit in the wrong hands can be quickly transformed into a worm or virus or used in an attack.

Memory Corruption Vulnerabilities

- WYSINWYX: What You See Is Not What You eXecute by G. Balakrishnan et. al.
  - Higher level code -> low-level representation
  - Seemingly separate variables -> contiguous memory addresses
- Contiguous memory locations allow for boundary violations!
- We need to peep into the memory layout to understand → ABI

Side effects

Side effects

- Over/underflow

Side effects

- Over/underflow
- Sensitive data corruption

Side effects

- Over/underflow
- Sensitive data corruption
- Control data corruption (control hijacking)

Side effects

- Over/underflow
- Sensitive data corruption
- Control data corruption (control hijacking)

If done properly-exploit

Side effects

- Over/underflow
- Sensitive data corruption
- Control data corruption (control hijacking)

If done properly-exploit

Otherwise crash!

Generic Approach to Vulnerability Detection

1. Find the weakness in the Software (sink)
2. Find the source of *tainted input* (source)
3. Find the tainted path from source to sink.

Generic Approach to Vulnerability Detection

1. Find the weakness in the Software (sink)
2. Find the source of *tainted input* (source)
3. Find the tainted path from source to sink.

You may have to do this at binary level

Mitigations

- Mitigation:
  - *Mitigations* are methods, techniques, processes, tools, or runtime libraries that can prevent or limit exploits against vulnerabilities.
  - At the code level, compiler assisted canary technique OR control flow integrity.
  - At a system or network level, a mitigation might involve turning off a port or filtering traffic to prevent an attacker from accessing a vulnerability OR OS provided techniques (ASLR, e.g.).

C and C++

- C and C++:
  - Popular programming languages.
  - Many vulnerabilities that have been reported to the CERT/CC have occurred in programs written in one of these two languages.
  - By design, these are *memory unsafe language.*

What is the Problem with C?

- Programmer errors
  - Failing to prevent writing beyond the boundaries of an array,
  - Failing to catch integer overflows and truncations,
- Lack of *type safety.*
  - Operations can legally act on signed and unsigned integers of differing lengths using implicit conversions and producing unrepresentable results.

Legacy Code

- A significant amount of legacy C code was created (and passed on) before the standardization of the language.
- Legacy C code is at higher risk for security flaws because of the looser compiler standards and is harder to secure because of the resulting coding style.

Motivating example

```
#include <stdio.h>
int get_cookie()
{
return rand();//random number
}
int main()
{
    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;
}
```

Motivating example conti..

Motivating example conti..

- What does it do?

Motivating example conti..

- What does it do?
- Does it do that as intended?

Motivating example conti..

- What does it do?
- Does it do that as intended?
- If not, why?

Motivating example conti..

- What does it do?
- Does it do that as intended?
- If not, why?
- Where is the problem?

Motivating example conti..

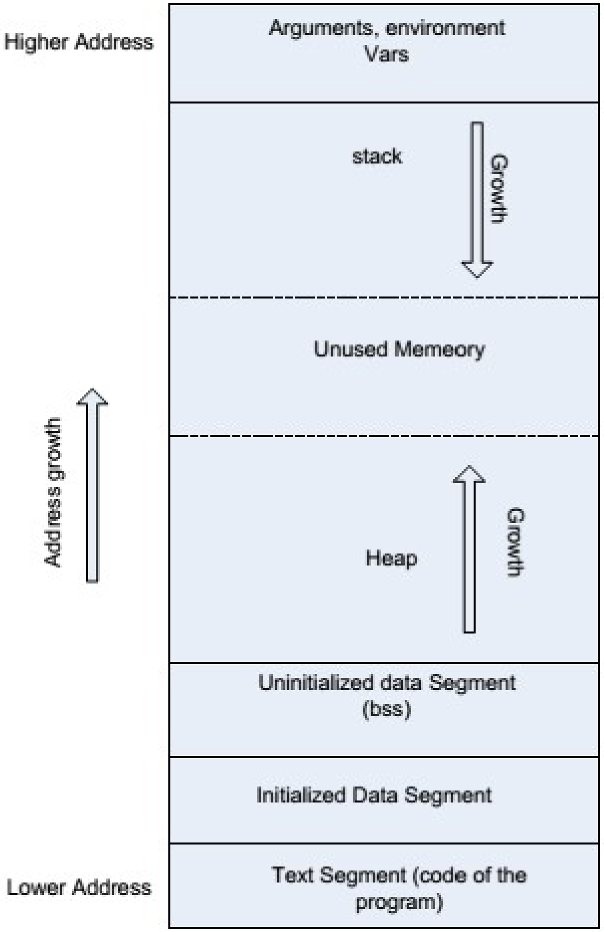
- What does it do?
- Does it do that as intended?
- If not, why?
- Where is the problem?
- We need to look stuff behind our program!!!

Motivating example conti..

- What does it do?
- Does it do that as intended?
- If not, why?
- Where is the problem?
- We need to look stuff behind our program!!!

Typical Memory Layout

- Windows PE and Linux ELF file formats
- Main segments of a compiled code:
  - The executable code (text segment)
  - The initialized data (data segment)
  - The uninitialized data (bss segment)
  - The heap (dynamic memory allocation)
  - The executable code and data of needed shared libraries (dynamically loaded into the space)
  - The program stack


Higher Address
Arguments, environment Vars
stack
Growth
Unused Memeory
Address growth
Heap
Growth
Uninitialized data Segment (bss)
Initialized Data Segment
Lower Address
Text Segment (code of the program)

x86 Assembly (32-bit)

- 6 general purpose registers
  - EAX, EBX, ECX, EDX, ESI EDI
- 2 special registers ESP, EBP
- 1 very special register EIP
- 100s of instructions
  - Data movement
  - Arithmetic
  - jump

Intel® 64 and IA-32 Architectures
Software Developer’s Manual
Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z

x86 Assembly (32-bit)

- 6 general purpose registers
  - EAX, EBX, ECX, EDX, ESI EDI
- 2 special registers ESP, EBP
- 1 very special register EIP
- 100s of instructions
  - Data movement
  - Arithmetic
  - jump

X86-64

%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp
%r8
%r9
%r10
%r11
%r12
%r13
%r14
%r15

Intel® 64 and IA-32 Architectures
Software Developer's Manual
Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z

Generic code syntax/semantic

- Two address mode
  - `Opcode operand1, operand2`
  - Intel- operand1 is destination, `mov rax, 10`
  - AT&T- operand2 is destination, `mov 10, rax`
- `Arithmetic`
  - `Add, sub, mul…` `add rax, rbx -> rax=rax+rbx`
- `Data movement`
  - `Mov, push pop…` `mov rax, 10 -> rax=10; mov eax, [ebx]`
- `Jump`
  - `Jmp <address>`

Calling conventions x86-32

- cdecl – parameters are pushes onto the stack, caller cleans the stack
- stdcall- parameters are pushes onto the stack, callee cleans the stack
- fastcall- first 3 parameters passed in EAX, EDX, ECX, rest on stack.
- thiscall- OOP, pointer to class object in ecx, rest are on stack.

Calling conventions x86-64

- Microsoft x64 calling convention
  - 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)

- Unix like systems
  - 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
  - Additional arguments are passed on the stack
  - The return value is stored in RAX and RDX

Calling conventions x86-64

- Microsoft x64 calling convention
  - 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)

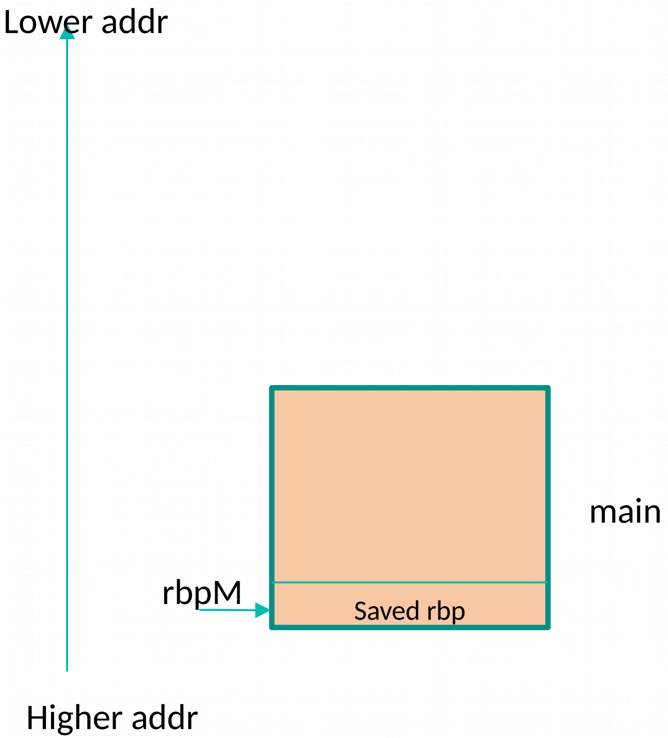
- Unix like systems
  - The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9
  - XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7 are used for certain floating point arguments
  - Additional arguments are passed on the stack
  - The return value is stored in RAX and RDX

If any of these argument registers are ***read first***, before writing, it is fastcall / x86-64 calling convention

Stack operation

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
movl    $0x6c726f57,-0x7(%rbp)
movw    $0x2164,-0x3(%rbp)
movb    $0x0,-0x1(%rbp)
mov     -0x20(%rbp),%rax
add     $0x8,%rax
mov     (%rax),%rax
mov     %rax,%rdi
callq   0x64a <buf_copy>
mov     $0x0,%eax
leaveq
retq
```

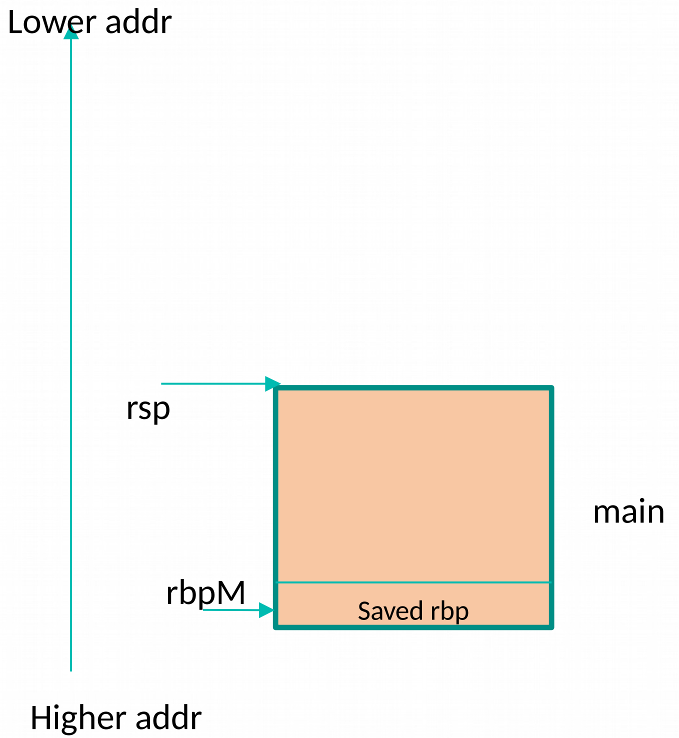
```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   0x520 <strcpy@plt>
mov     $0x0,%eax
leaveq
retq
```



Stack operation

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
movl    $0x6c726f57,-0x7(%rbp)
movw    $0x2164,-0x3(%rbp)
movb    $0x0,-0x1(%rbp)
mov     -0x20(%rbp),%rax
add     $0x8,%rax
mov     (%rax),%rax
mov     %rax,%rdi
callq   0x64a <buf_copy>
mov     $0x0,%eax
leaveq
retq
```

```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   0x520 <strcpy@plt>
mov     $0x0,%eax
leaveq
retq
```

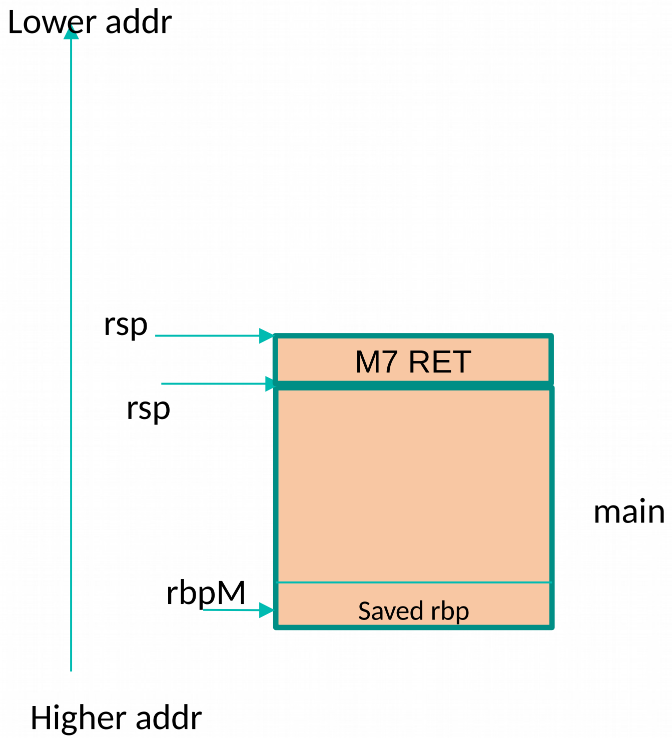


bristol

Stack operation

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
movl    $0x6c726f57,-0x7(%rbp)
movw    $0x2164,-0x3(%rbp)
movb    $0x0,-0x1(%rbp)
mov     -0x20(%rbp),%rax
add     $0x8,%rax
mov     (%rax),%rax
mov     %rax,%rdi
callq   0x64a <buf_copy>
mov     $0x0,%eax
leaveq
retq
```

```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   0x520 <strcpy@plt>
mov     $0x0,%eax
leaveq
retq
```

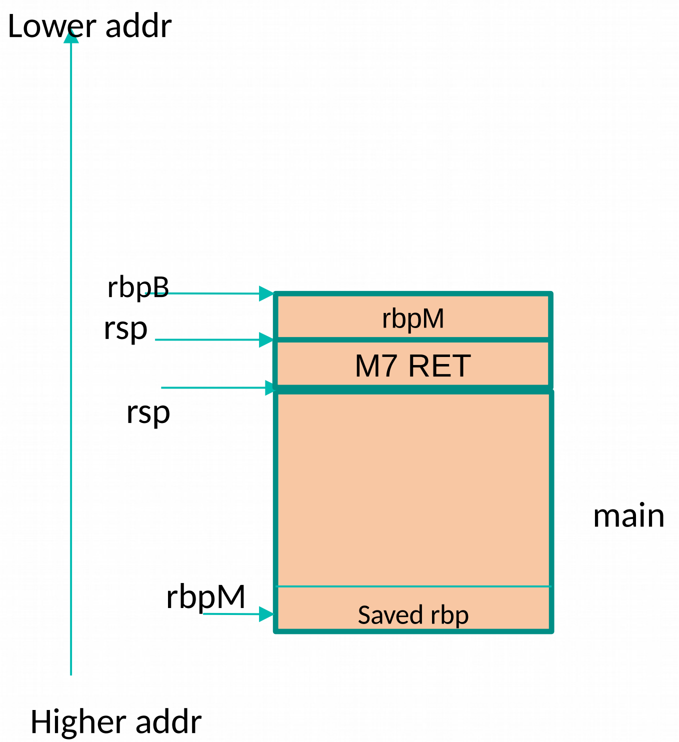


brist

Stack operation

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
movl    $0x6c726f57,-0x7(%rbp)
movw    $0x2164,-0x3(%rbp)
movb    $0x0,-0x1(%rbp)
mov     -0x20(%rbp),%rax
add     $0x8,%rax
mov     (%rax),%rax
mov     %rax,%rdi
callq   0x64a <buf_copy>
mov     $0x0,%eax
leaveq
retq
```

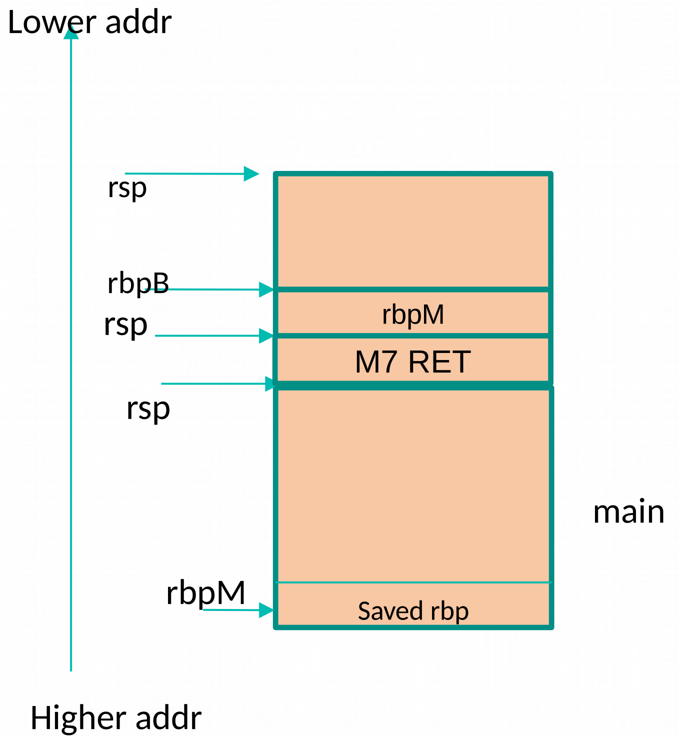
```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   0x520 <strcpy@plt>
mov     $0x0,%eax
leaveq
retq
```



Stack operation

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
movl    $0x6c726f57,-0x7(%rbp)
movw    $0x2164,-0x3(%rbp)
movb    $0x0,-0x1(%rbp)
mov     -0x20(%rbp),%rax
add     $0x8,%rax
mov     (%rax),%rax
mov     %rax,%rdi
callq   0x64a <buf_copy>
mov     $0x0,%eax
leaveq
retq
```

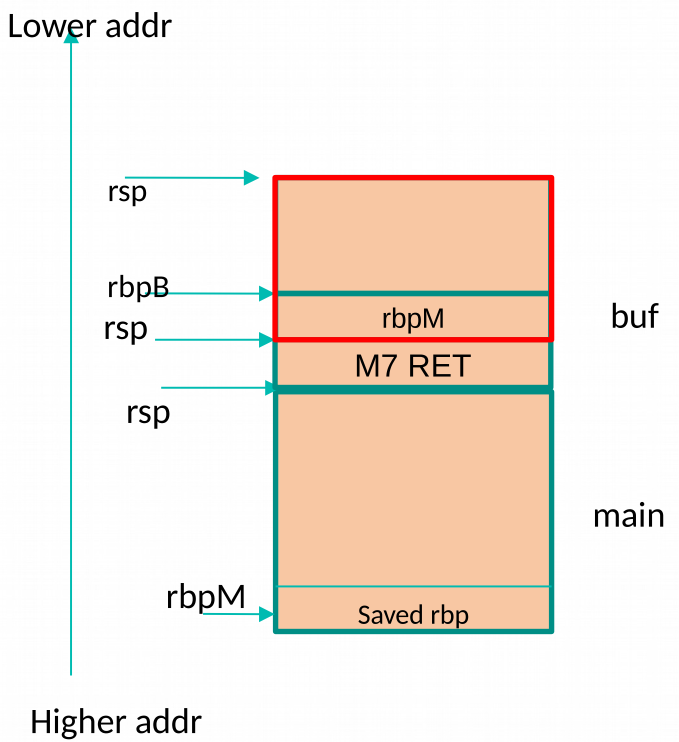
```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   0x520 <strcpy@plt>
mov     $0x0,%eax
leaveq
retq
```



Stack operation

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
movl    $0x6c726f57,-0x7(%rbp)
movw    $0x2164,-0x3(%rbp)
movb    $0x0,-0x1(%rbp)
mov     -0x20(%rbp),%rax
add     $0x8,%rax
mov     (%rax),%rax
mov     %rax,%rdi
callq   0x64a <buf_copy>
mov     $0x0,%eax
leaveq
retq
```

```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   0x520 <strcpy@plt>
mov     $0x0,%eax
leaveq
retq
```



Example: C to assembly

```c
int main(int argc, char *argv[]) {
        char name[]="World!";
        buf_copy(argv[1]);
                return 1;
                }
```

```c
int buf_copy(char *string) {
        char buffer[20];
      strcpy(buffer, string);
            return 1;
              }
```

```
push    %rbp
mov     %rsp,%rbp
sub     $0x20,%rsp
mov     %edi,-0x14(%rbp)
mov     %rsi,-0x20(%rbp)
movl    $0x6c726f57,-0x7(%rbp)
movw    $0x2164,-0x3(%rbp)
movb    $0x0,-0x1(%rbp)
mov     -0x20(%rbp),%rax
add     $0x8,%rax
mov     (%rax),%rax
mov     %rax,%rdi
callq   0x64a <buf_copy>
mov     $0x0,%eax
leaveq
retq
```

```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     -0x28(%rbp),%rdx
lea     -0x20(%rbp),%rax
mov     %rdx,%rsi
mov     %rax,%rdi
callq   0x520 <strcpy@plt>
mov     $0x0,%eax
leaveq
retq
```

Know your Tools

- Disassembler/debugger
  - Windows: OllyDbg, PyDbg, Immunity Debugger, IDA Pro
  - Linux: GDB ▮▮ , Evan's Debugger (EDB), IDA Pro
- Language: Your choice, Python (my choice)
- Hex viewer: HxD HexEditor

compiling options

Useful

- GCC:
  - gcc -fno-stack-protector -z execstack vulnerable.c
    - -fno-stack-protector disables SSP (stack guard)
    - -z execstack marks the stack as executable
  - -mno-accumulate-outgoing-args -mpush-args
    - ## forcing GCC to push the arguments on the stack

- Windows CL
- /GS-

Coming back to vulnerability

Coming back to vulnerability

- Finding vulnerability is first thing.
  - Find it and Patch it OR
  - How to exploit it?
  - Hijacking the control -> controlling which instruction(s) will be executed next.
  - Candidates: saved return addr, function pointers, SEH etc.