(Early) Memory Corruption Attacks (cont'd)

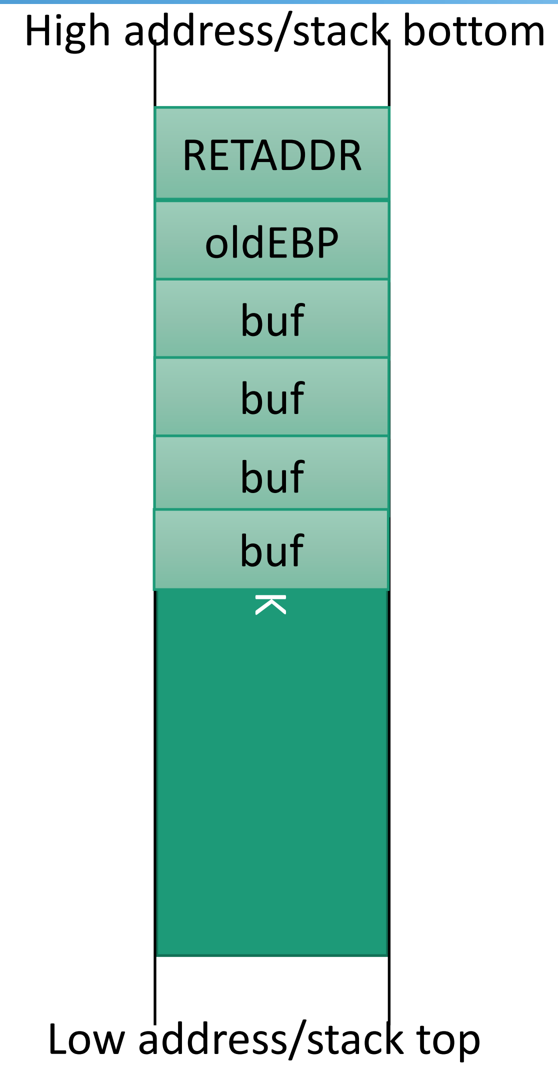
CS-576 Systems Security

Instructor: Georgios Portokalidis

Fall 2018

Stack Overflow with FP

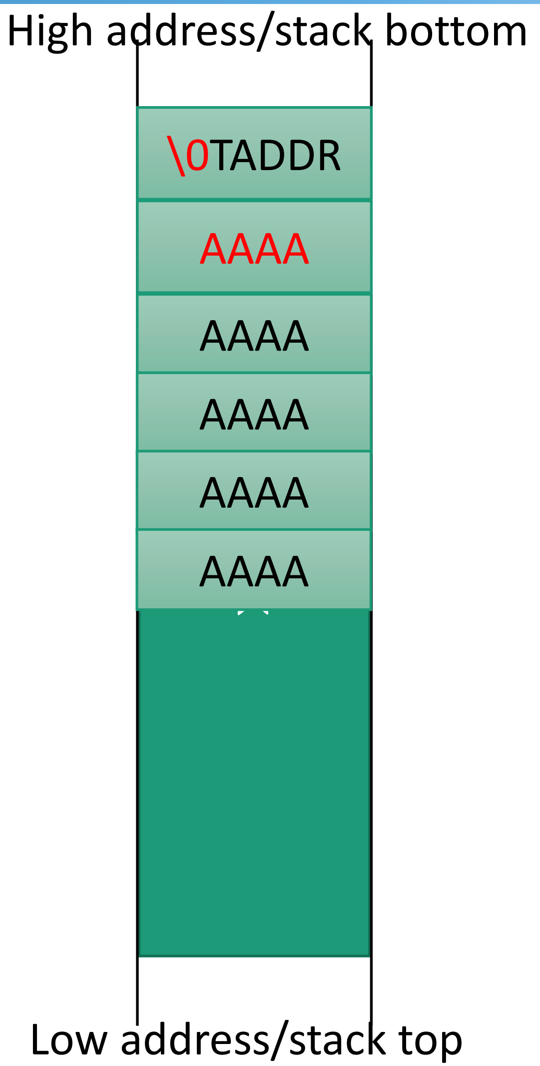
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```



Stack Overflow with FP

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

./mytest AAAAAAAAAAAAAAAAAAAA

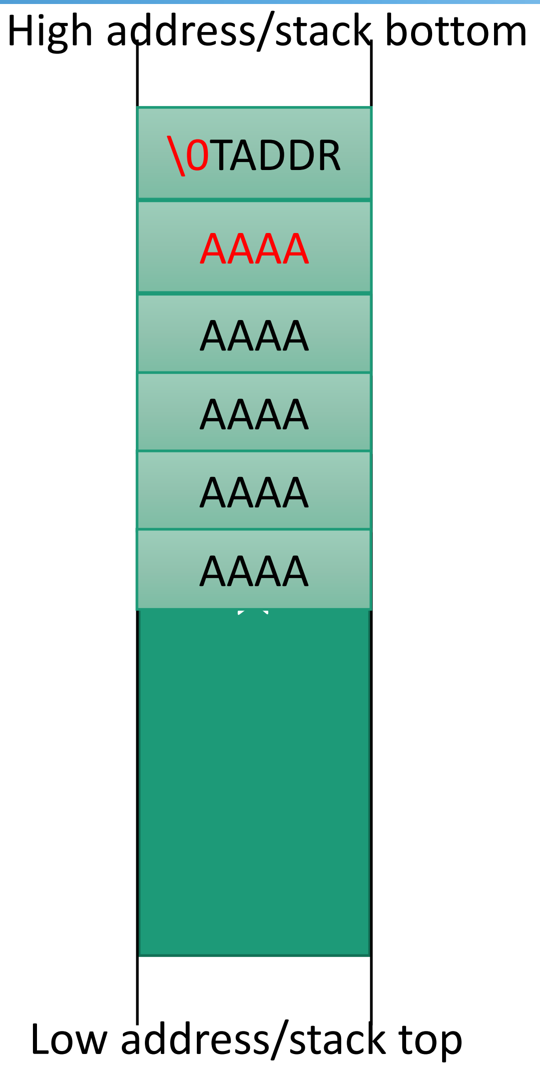


Stack Overflow with FP

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

./mytest AAAAAAAAAAAAAAAAAAAA

```
80484e1: c9                    leave
80484e2: c3                    ret
```



Stack Overflow with FP

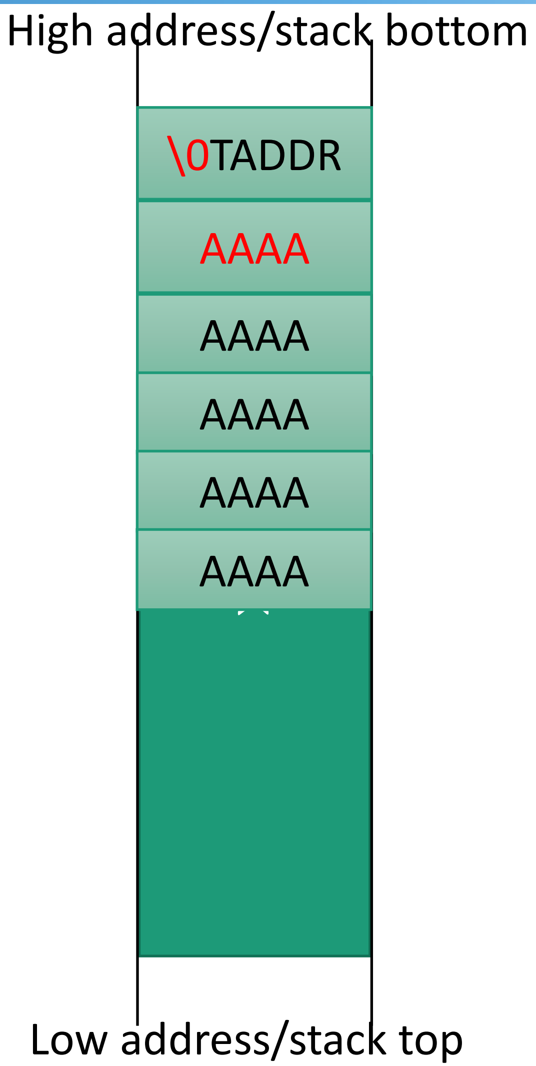
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

./mytest AAAAAAAAAAAAAAAAAAA

Function exit (LEAVE)

```
movl    %ebp, %esp
pop     %ebp
```

8048 ... ve
8048



Stack Overflow with FP

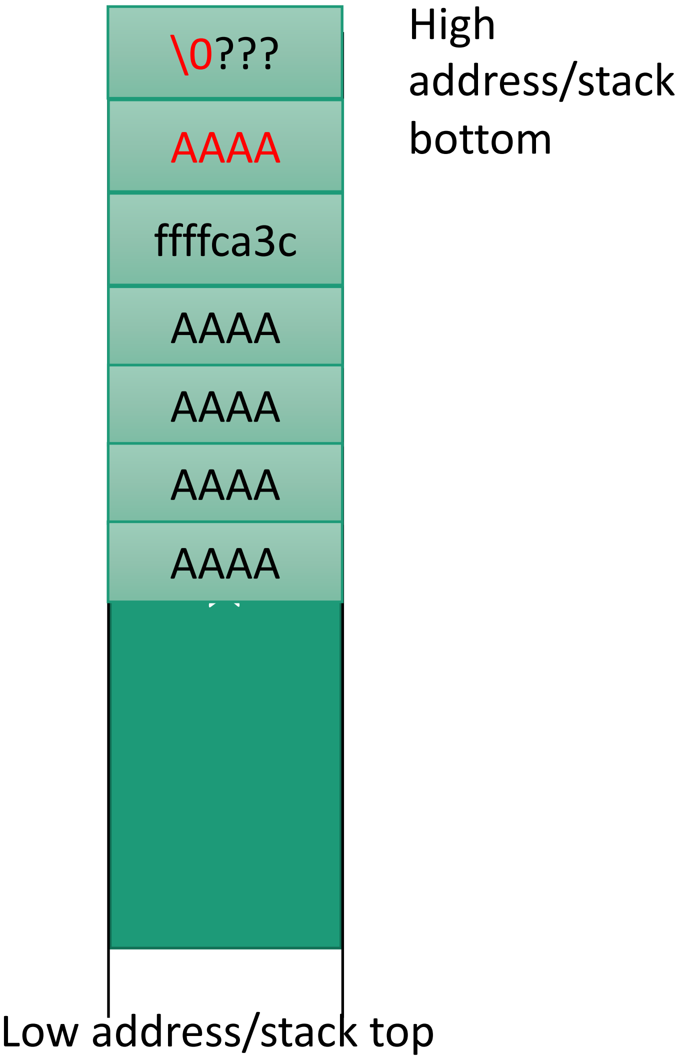
```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

./mytest AAAAAAAAAAAAAAA\x3c\xca\xff\xffAAAA

```
8048                                    ve
8048
```

Function exit (LEAVE)

```
movl      %ebp, %esp
pop       %ebp
```



Writing Shellcode

How to Write Shellcode

Code in assembly → compile with GCC → Binary code

Compile assembly program to object file

```
gcc -c shellcode.S
```

View generated code

```
objdump –d shellcode.o
```

Copy text segment to separate file

```
objcopy -O binary --only-section=.text shellcode.o shellcode.sc
```

Usually encode binary code as text in C, perl, python, etc.

```
hexdump -v -e '"\\""x" 1/1 "%02x" ""' shellcode.sc
```

Calling Functions

Shellcode can call functions loaded in the address space

- Assuming you know their offset from the call instruction

Example:

Addr0: call AddrF-Addr1

Addr1: ins

<function>:

AddrF: …

Calling System Calls

Shellcode can call systems calls

Example:

Addr0: syscall

Linux:

- System call API is powerful, easy to use, and well documented

Windows

- System call API is harder to use and not well documented

Calling System Calls

Shellcode can call systems calls

Example:

Addr0: syscall

Linux:

- **System call API is powerful, easy to use, and well documented**

Windows

- System call API is harder to use and not well documented

Hello World Shellcode

Write “Hello World\n” to standard output

Gracefully terminate program

Hello World Shellcode

Write "Hello World\n" to standard output

- Use write() system call

Gracefully terminate program

- Use exit() system call

Linux System Call Conventions

The kernel interface uses %rdi, %rsi, %rdx, %r10, %r8 and %r9 for passing arguments

A system-call is done via the syscall instruction. The kernel destroys registers %rcx and %r11

The number of the syscall has to be passed in register %rax

System-calls are limited to six arguments, no argument is passed directly on the stack

Returning from the syscall, register %rax contains the result of the system-call. A value in the range between -4095 and -1 indicates an error, it is -errno

Linux System Call Table

https://syscalls.kernelgrok.com/

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
4	sys_stat	const char *filename	struct stat *statbuf				
5	sys_fstat	unsigned int fd	struct stat *statbuf				
6	sys_lstat	fconst char *filename	struct stat *statbuf				
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs			
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin			

Calling write()

Find the API for sys_write()

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			

write(1, “Hello World\n”, 11);

- 1 → file descriptor corresponding to **stdout**
- “Hello World\n” → Pointer to data to be written
- 11 → Number of bytes to be written

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout

        mov     $12, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write
```

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                 # system call 1 is write
        mov     $1, %rdi                 # file handle 1 is stdout

        mov     $12, %rdx                # number of bytes
        syscall                          # invoke operating system to do the write

message:
        .ascii  "Hello world\n"
```

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi
        mov     $12, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

message:
        .ascii  "Hello world\n"
```

Calling exit()

Find the API for sys_exit()

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
60	sys_exit	int error_code					
61	sys_wait4	pid_t upid	int *stat_addr	int options	struct rusage *ru		

exit(0);

- 0 → return value for correct termination

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi
        mov     $12, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello world\n"
```

Example Shellcode

```
# write(1, message, 12)
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi
        mov     $12, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     $60, %rax
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello world\n"
```

xor reg, reg
sub reg, reg

Common idiom on x86 for zeroing a register

Compiling Shellcode

```
gcc -c hello.S
ld -o hello hello.o
```

Binary Code

objdump –d hello.o

```
    0:   48 c7 c0 01 00 00 00     mov    $0x1,%rax
    7:   48 c7 c7 01 00 00 00     mov    $0x1,%rdi
    e:   48 c7 c6 00 00 00 00     mov    $0x0,%rsi
   15:   48 c7 c2 0d 00 00 00     mov    $0xc,%rdx
   1c:   0f 05                    syscall
   1e:   48 c7 c0 3c 00 00 00     mov    $0x3c,%rax
   25:   48 31 ff                 xor    %rdi,%rdi
   28:   0f 05                    syscall

000000000000002a <message>:
   2a:   48                       rex.W
   2b:   65 6c                    gs insb (%dx),%es:(%rdi)
   2d:   6c                       insb   (%dx),%es:(%rdi)
   2e:   6f                       outsl  %ds:(%rsi),(%dx)
   2f:   20 77 6f                 and    %dh,0x6f(%rdi)
   32:   72 6c                    jb     a0 <message+0x76>
   34:   64                       fs
   35:   0a                       .byte 0xa
```

Object Code

objdump –d hello.o

```
   0:    48 c7 c0 01 00 00 00       mov     $0x1,%rax
   7:    48 c7 c7 01 00 00 00       mov     $0x1,%rdi
   e:    48 c7 c6 00 00 00 00       mov     $0x0,%rsi
  15:    48 c7 c2 0d 00 00 00       mov     $0xc,%rdx
  1c:    0f 05                      syscall
  1e:    48 c7 c0 3c 00 00 00       mov     $0x3c,%rax
  25:    48 31 ff                   xor     %rdi,%rdi
  28:    0f 05                      syscall

000000000000002a <message>:
  2a:    48                         rex.W
  2b:    65 6c                      gs insb (%dx),%es:(%rdi)
  2d:    6c                         insb    (%dx),%es:(%rdi)
  2e:    6f                         outsl   %ds:(%rsi),(%dx)
  2f:    20 77 6f                   and     %dh,0x6f(%rdi)
  32:    72 6c                      jb      a0 <message+0x76>
  34:    64                         fs
  35:    0a                         .byte 0xa
```

Linked Code

objdump –d hello

```
0000000000400078 <_start>:
  400078:       48 c7 c0 01 00 00 00      mov    $0x1,%rax
  40007f:       48 c7 c7 01 00 00 00      mov    $0x1,%rdi
  400086:       48 c7 c6 a2 00 40 00      mov    $0x4000a2,%rsi
  40008d:       48 c7 c2 0c 00 00 00      mov    $0xc,%rdx
  400094:       0f 05                     syscall
  400096:       48 c7 c0 3c 00 00 00      mov    $0x3c,%rax
  40009d:       48 31 ff                  xor    %rdi,%rdi
  4000a0:       0f 05                     syscall

00000000004000a2 <message>:
  4000a2:       48                        rex.W
  4000a3:       65 6c                     gs insb (%dx),%es:(%rdi)
  4000a5:       6c                        insb   (%dx),%es:(%rdi)
  4000a6:       6f                        outsl  %ds:(%rsi),(%dx)
  4000a7:       20 77 6f                  and    %dh,0x6f(%rdi)
  4000aa:       72 6c                     jb     400118 <message+0x76>
  4000ac:       64                        fs
  4000ad:       0a                        .byte 0xa
```

Getting the Shellcode

```
objcopy -O binary --only-section=.text hello hello.sc
```

```
echo -n "const char shellcode[] = \"" > hello.c
hexdump -v -e '"\\""x" 1/1 "%02x" ""' hello.sc >> hello.c
echo '";' >> hello.c
```

```
const char shellcode[] =
"\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\xc7\xc6\xa2\x00
\x40\x00\x48\xc7\xc2\x0c\x00\x00\x00\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00\x48\
x31\xff\x0f\x05\x48\x65\x6c\x6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a";
```

Using the Shellcode

```
const char shellcode[] =
"\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48
\xc7\xc6\xa2\x00\x40\x00\x48\xc7\xc2\x0c\x00\x00\x00\x0f\x05\
x48\xc7\xc0\x3c\x00\x00\x00\x48\x31\xff\x0f\x05\x48\x65\x6c\x
6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a";
```

Shellcode can be written to stdout

- write(1, shellcode, sizeof(shellcode));

How could you execute it from within a C program?

"Special" Bytes Limitations

```
const char shellcode[] =
"\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48
\xc7\xc6\xa2\x00\x40\x00\x48\xc7\xc2\x0c\x00\x00\x00\x0f\x05\
x48\xc7\xc0\x3c\x00\x00\x00\x48\x31\xff\x0f\x05\x48\x65\x6c\x
6c\x6f\x20\x77\x6f\x72\x6c\x64\x0a";
```

Certain characters may not be allowed

- strcpy() stops copying at null byte
- gets() reads one line at a time
- Input may need to be alphanumeric

Bypasses:

- Rewrite shellcode to avoid characters
- Encode shellcode

Eliminating 0 Bytes

Zero in opcodes

- Alternate instructions can achieve a similar result

Zero in constants

- Use multiple instructions to construct constants

Eliminating 0 Bytes

Zero in opcodes

- Alternate instructions can achieve a similar result

Zero in constants

- Use multiple instructions to construct constants

```
0:48 31 c0          xor    %rax,%rax
3:48 ff c0          inc    %rax
```

Eliminating 0 Bytes

```
        # write(1, message, 12)
        xor    %rax, %rax
        inc    %rax
        #mov    $1, %rax                     # system call 1 is write
        xor    %rdi, %rdi
        inc    %rdi
        #mov    $1, %rdi                     # file handle 1 is stdout
        mov    $message, %rsi
        xor    %rdx, %rdx
        addb    $12, %dl
        #mov    $12, %rdx                   # number of bytes
        syscall                              # invoke operating system to do the write

        # exit(0)
        xor    %rax, %rax
        addb    $60, %al
        #xor   $60, %rax                     # system call 60 is exit
        xor    %rdi, %rdi                   # we want return code 0
        syscall                              # invoke operating system to exit
message:
        .ascii "Hello world\n"
```

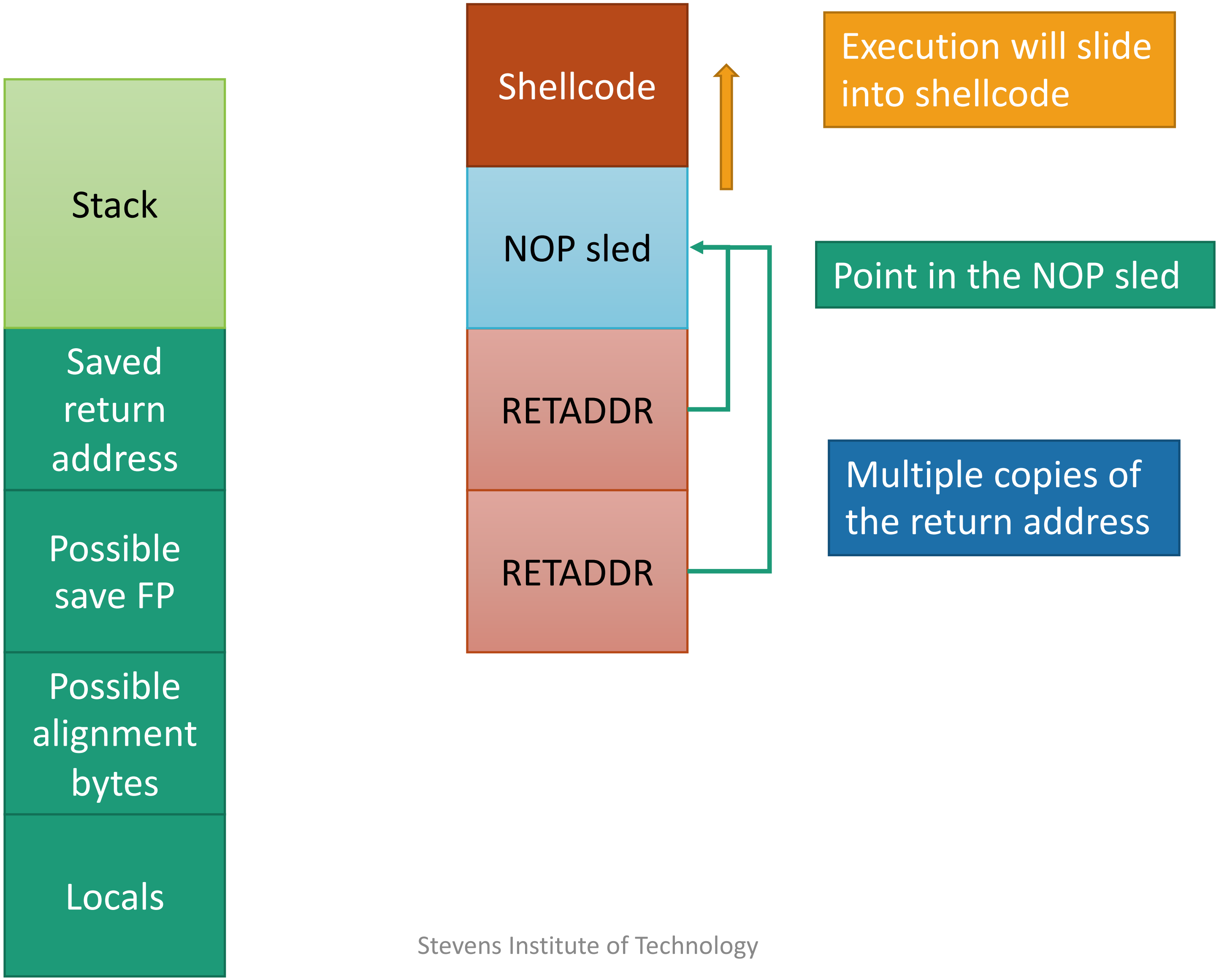
Using RIP-Relative Addressing

```
        # write(1, message, 13)
        xor    %rax, %rax
        inc    %rax
        #mov    $1, %rax                # system call 1 is write
        xor    %rdi, %rdi
        inc    %rdi
        #mov    $1, %rdi                # file handle 1 is stdout
        lea    message(%rip), %rsi     # rip relative load of message address
        xor    %rdx, %rdx
        addb    $13, %dl
        #mov    $13, %rdx              # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        xor    %rax, %rax
        addb    $60, %al
        #xor   $60, %rax                # system call 60 is exit
        xor    %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii "Hello world\n"
```

Eliminating 0 Bytes

```
        # write(1, message, 12)
        xor    %rax, %rax
        inc    %rax
        #mov    $1, %rax                    # system call 1 is write
        xor    %rdi, %rdi
        inc    %rdi
        #mov    $1, %rdi                    # file handle 1 is stdout
        #lea    message(%rip), %rsi
        lea     0x01111129(%rip), %rsi       # address of string to output
        sub    $0x01111110, %rsi
        xor    %rdx, %rdx
        addb    $12, %dl
        #mov    $12, %rdx                  # number of bytes
        syscall                             # invoke operating system to do the write

        # exit(0)
        xor    %rax, %rax
        addb    $60, %al
        #xor   $60, %rax                    # system call 60 is exit
        xor    %rdi, %rdi                  # we want return code 0
        syscall                             # invoke operating system to exit
message:
        .ascii "Hello world\n"
```

Making Exploits More Generic



Non-Control Data Attacks

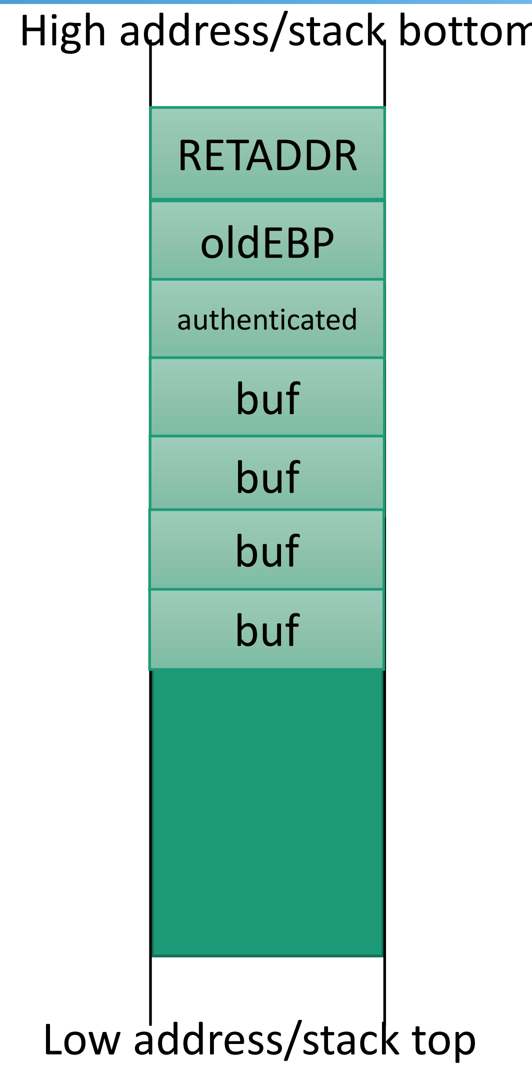
Attacks overwriting data not directly used in control flow

Essentially corrupting program state that affects its security

- For example: Disabling/Bypassing a security mechanism

Example

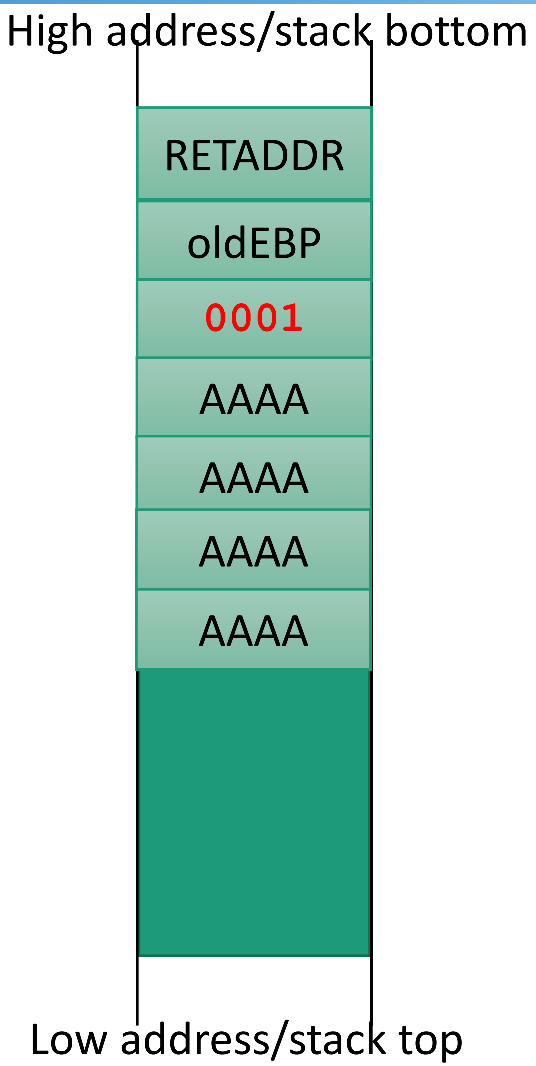
```
static int mytest(char *str)
{
        int authenticated = 0;
        char buf[16];

        read(STDIN_FILENO, buf, 32);
        if (check_pass(buf))
                authenticated = 1;

        do_something(authenticated);
}
```



Example

```
static int mytest(char *str)
{
          int authenticated = 0;
          char buf[16];

          read(STDIN_FILENO, buf, 32);
          if (check_pass(buf))
                    authenticated = 1;

          do_something(authenticated);
}
```

./mytest AAAAAAAAAAAAAAAA\x01\x00\x00\x00



Heap Overflows

Heap Overflows

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

int main(int argc, char *argv[]) {
    FILE = *filed;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    strcpy(outputfile, "/tmp/foobar");
    strcpy(userinput, argv[1]);

    filed = fopen(outputfile, "a");
    if(filed == NULL){
        fprintf(stderr, "error opening file %s\n", outputfile);
        exit(1);     }
    fprintf(filed, "%s\n", userinput);
    fclose(filed);
    return 0;
}
```

Heap Structure

```
char *userinput = malloc(20);
char *outputfile = malloc(20);
```

Overwriting Program Data

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

int main(int argc, char *argv[]) {
    FILE = *filed;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    strcpy(outputfile, "/tmp/foobar");
    strcpy(userinput, argv[1]);

    filed = fopen(outputfile, "a");
    if(filed == NULL){
        fprintf(stderr, "error opening file %s\n", outputfile);
        exit(1);     }
    fprintf(filed, "%s\n", userinput);
    fclose(filed);
    return 0;
}
```

Overwrite outputfile

Overwriting Program Data

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

int main(int argc, char *argv[]) {
    FILE = *filed;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    strcpy(outputfile, "/tmp/foobar
    strcpy(userinput, argv[1]);

    filed = fopen(outputfile, "a");
    if(filed == NULL){
        fprintf(stderr, "error opening file %s\n", outputfile);
        exit(1);     }
    fprintf(filed, "%s\n", userinput);
    fclose(filed);
    return 0;
}
```

Control what file is written to

Overwriting Program Data

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

int main(int argc, char *argv[]) {
    FILE = *filed;
    char *userinput = malloc(20);
    char *outputfile = malloc(20);

    strcpy(outputfile, "/tmp/foobar");
    strcpy(userinput, argv[1]);

    filed = fopen(outputfile, "a");
    if(filed == NULL){
        fprintf(stderr, "error opening file %s\n",
        exit(1);     }
    fprintf(filed, "%s\n", userinput);
    fclose(filed);
    return 0;
}
```

Whether you can directly control a code pointer depends on the program

What are good targets?

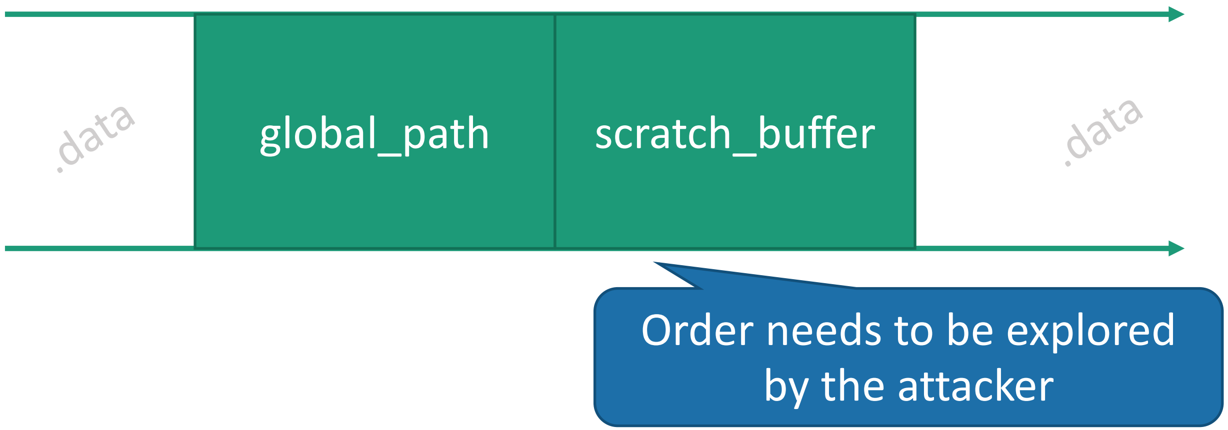
Append to that file

Global Data Overflows

Global Data Overflow

Arrays in .bss and .data segments

```
static char global_path[256];
static char scratch_buffer[1024];


int main(int argc, char **argv)
{
```



Integer Overflows

Integer Overflows

Integers wrap around!

Can be used to bypass if statements

Can do arbitrary writes by referencing negative offsets in arrays

```
buf[-1000] = input
```

```c
/* width1.c - exploiting a trivial widthness bug */
   #include <stdio.h>
   #include <string.h>

   int main(int argc, char *argv[]){
           unsigned short s;
           int i;
           char buf[80];

           if(argc < 3){
                   return -1;
           }

           i = atoi(argv[1]);
           s = i;

           if(s >= 80){            /* [w1] */
                   printf("Oh no you don't!\n");
                   return -1;
           }

           printf("s = %d\n", s);

           memcpy(buf, argv[2], i);
           buf[i] = '\0';
           printf("%s\n", buf);

           return 0;
   }
```

```
/* width1.c - exploiting a trivial widthness bug */
   #include <stdio.h>
   #include <string.h>

   int main(int argc, char *argv[]){
           unsigned short s;
           int i;
           char buf[80];

           if(argc < 3){
                   return -1;
           }

           i = atoi(argv[1]);
           s = i;

           if(s >= 80){            /* [w1] */
                   printf("Oh no you don't!\n");
                   return -1;
           }

           printf("s = %d\n", s);

           memcpy(buf, argv[2], i);
           buf[i] = '\0';
           printf("%s\n", buf);

           return 0;
   }
```

Use-After-Free Vulnerabilities

A buffer, object, etc. is used after being freed

Scenario:

1. Program allocates and then later frees block A
2. Attacker allocates block B, reusing the memory previously allocated to block A
3. Attacker writes data into block B
4. Program uses freed block A, accessing the data the attacker left there

```
int main(int argc, char **argv)
{
    struct objectA *objA;
    struct objectB *objB;

    objA = malloc(sizeof(struct object A));
    funcA(objA);  /* frees objA */
    objB = malloc(sizeof(struct object B));
   funcB(objhB) /* writes on objB */
   …
  funcAA(objA); /*accesses freed objA */
```

Use-After-Free Vulnerabilities

A buffer, object, etc. is used after being freed

Scenario:

1. Program... later fre...
2. Attacker... reusing the memory... previous... block A
3. Attacker... block B
4. Program... A, acces... attacker left there

```
struct objectA {
        …       …
        void (*fprt)();
        char *string;
        …
 }
```

```
struct objectB {
        …       …
        int a;
        long b;
        …
 }
```

```
int main(int argc, char **argv)
{
    ... *objA;
    ... *objB;
    ... sizeof(struct object A));
    funcA(objA); /* frees objA */
    ... sizeof(struct object B));
    ... writes on objB */
    ... /*accesses freed objA */
```

C++ Vulnerabilities

```
class ClassA {
...
virtual void vfunc1() { /* code Avf1 */
void func1() { /* code Af1 */
};
```

```
class ClassB : ClassA {
...
virtual void vfunc1() { /* code Bvf1 */
virtual void vfunc2() { /* code Bvf2 */
void func2() { /* code Bf2 */ }
};
```

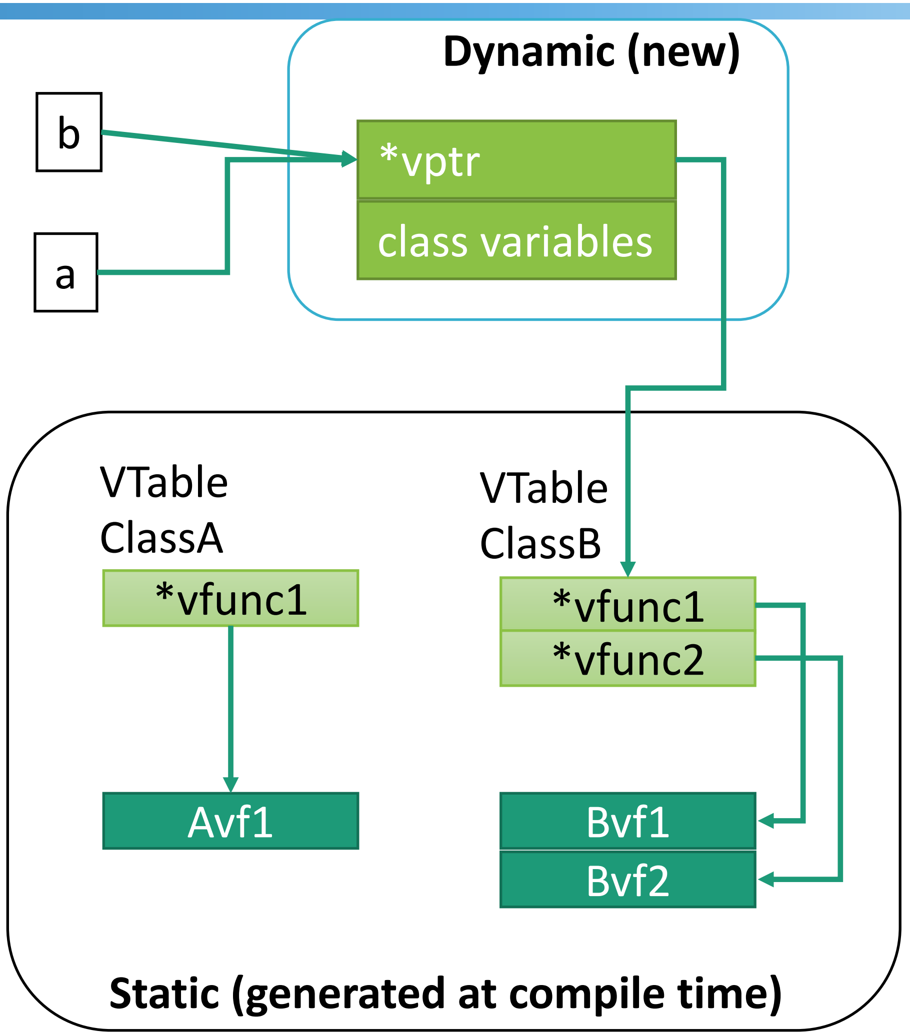
```
int main(int argc, char **argv)
{
    ClassA *a;
    ClassB *b;

    b = new ClassB();
    ....
    a = b;
    a->vfunc1();
    b->vfunc1();
```

Which functions are called?

Late Binding and VTables

The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation

VTables are used to enable late binding

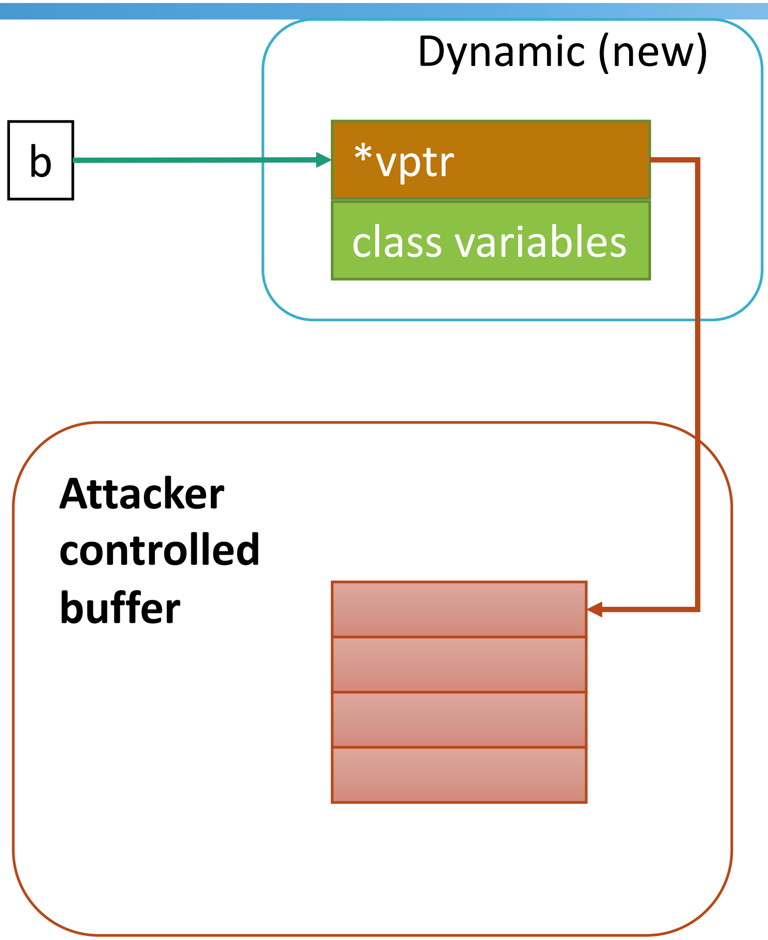


Late Binding and VTables

The actual virtual function that will be called depends on the object type NOT on the class type of the variable used in the invocation

VTables are used to enable late binding

Heap overflows can be used to corrupt the vptr



Type Confusion

Type Confusion

```
class ClassA {
…
virtual void vfunc1() { /* code Avf1 */
void func1() { /* code Af1 */
};
```

```
class ClassB {
…
virtual void foobar(int foo, int bar);
}
```

```
int main(int argc, char **argv)
{
    ClassA *a;
    ClassB *b;

    a= new ClassA();
    ….
    b = (Class B)objA;

    b->foobar();
```

C/C++ is weakly typed

Type Confusion is "In"

One Perfect Bug: Exploiting Type Confusion in Flash

- https://googleprojectzero.blogspot.com/2015/07/one-perfect-bug-exploiting-type_20.html

CVE-2016-3185 php: Type confusion vulnerability in make_http_soap_request()

- https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2016-3185

Python xmlparse_setattro() Type Confusion

- http://bugs.python.org/issue25019

Exploiting Type Confusion Vulnerabilities in Oracle JRE (CVE-2011-3521/CVE-2012-0507)

- http://schierlm.users.sourceforge.net/TypeConfusion.html