CIS551: Computer and Network Security

Jonathan M. Smith

jms@cis.upenn.edu

01/27/2014

Stuxnet

- Did you understand the paper???
- Very sophisticated system
  - Multiple “zero-day” vulnerabilities exploited
  - Programmable logic controller target
  - Transport via Internet + thumb drive(!)
  - Various hiding techniques
  - Uses stolen cryptographic material
  - Lots of moving parts

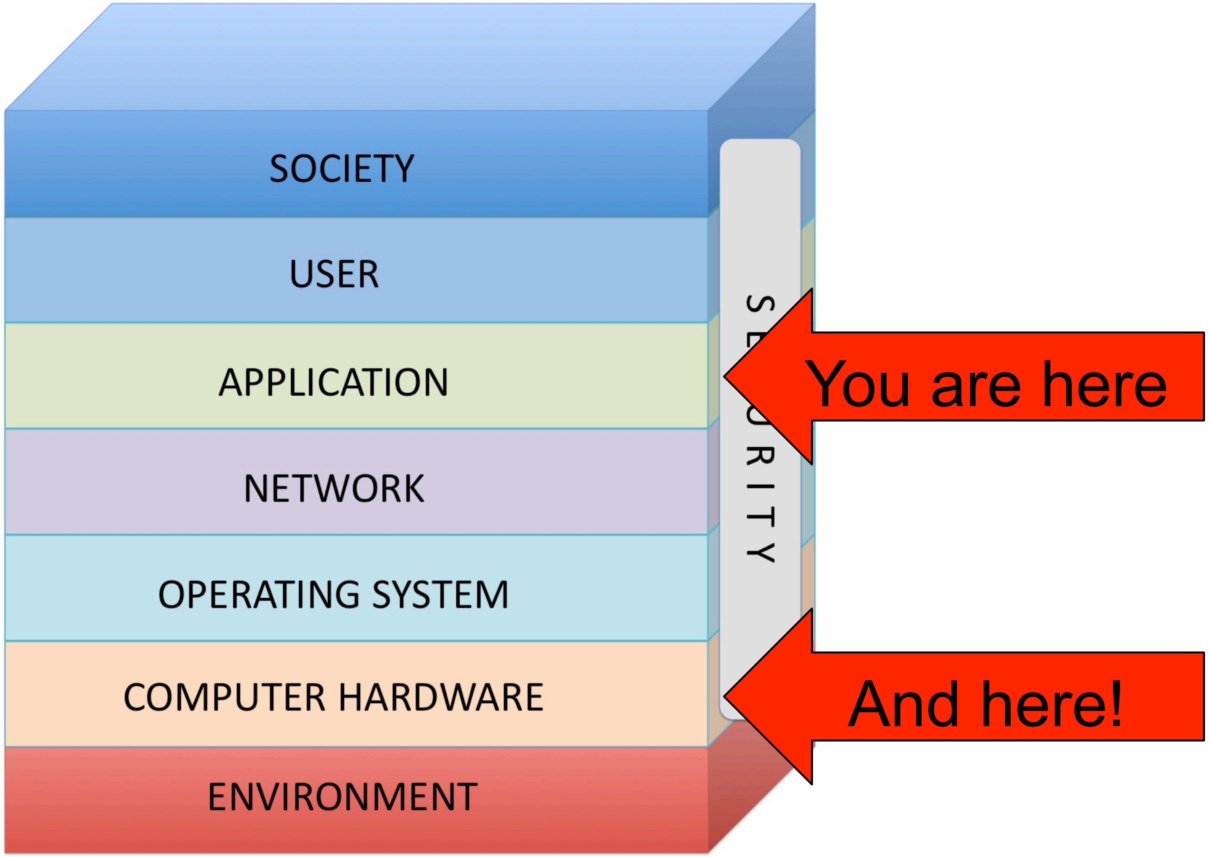
Stuxnet: Lots of moving parts

- Many software subsystems
  - Windows versions
  - Siemens software
- Networking
- Hardware devices
- Bad user habits

CIS551 Topics

- Computer Security
  - Software/Languages, Computer Arch.
  - Access Control, Operating Systems
  - Threats: Vulnerabilities, Viruses
- Computer Networks
  - Physical layers, Internet, WWW, Applications
  - Cryptography in several forms
  - Threats: Confidentiality, Integrity, Availability
- Systems Viewpoint
  - Users, social engineering, insider threats

Sincoskie NIS model



W.D. Sincoskie, *et al.* “Layer Dissonance and Closure in Networked Information Security” (white paper)

Software Security

- Software makes a Turing Machine useful
- Market pressures, programmer discipline, language, tools and social acceptance all allow buggy software
- Security is doing the right thing for the right person at the right place at the right time – nothing more or less

Bugs, vulnerabilities, exploits

- A **bug** is incorrect code
- Some bugs create **vulnerabilities**
- Some vulnerabilities are discovered and exploited – these are **exploits**
  - A subset of these are publicized / “known”
  - Non-public: “zero-day” (no signatures, etc.)
- Some bugs are *amazingly* stupid

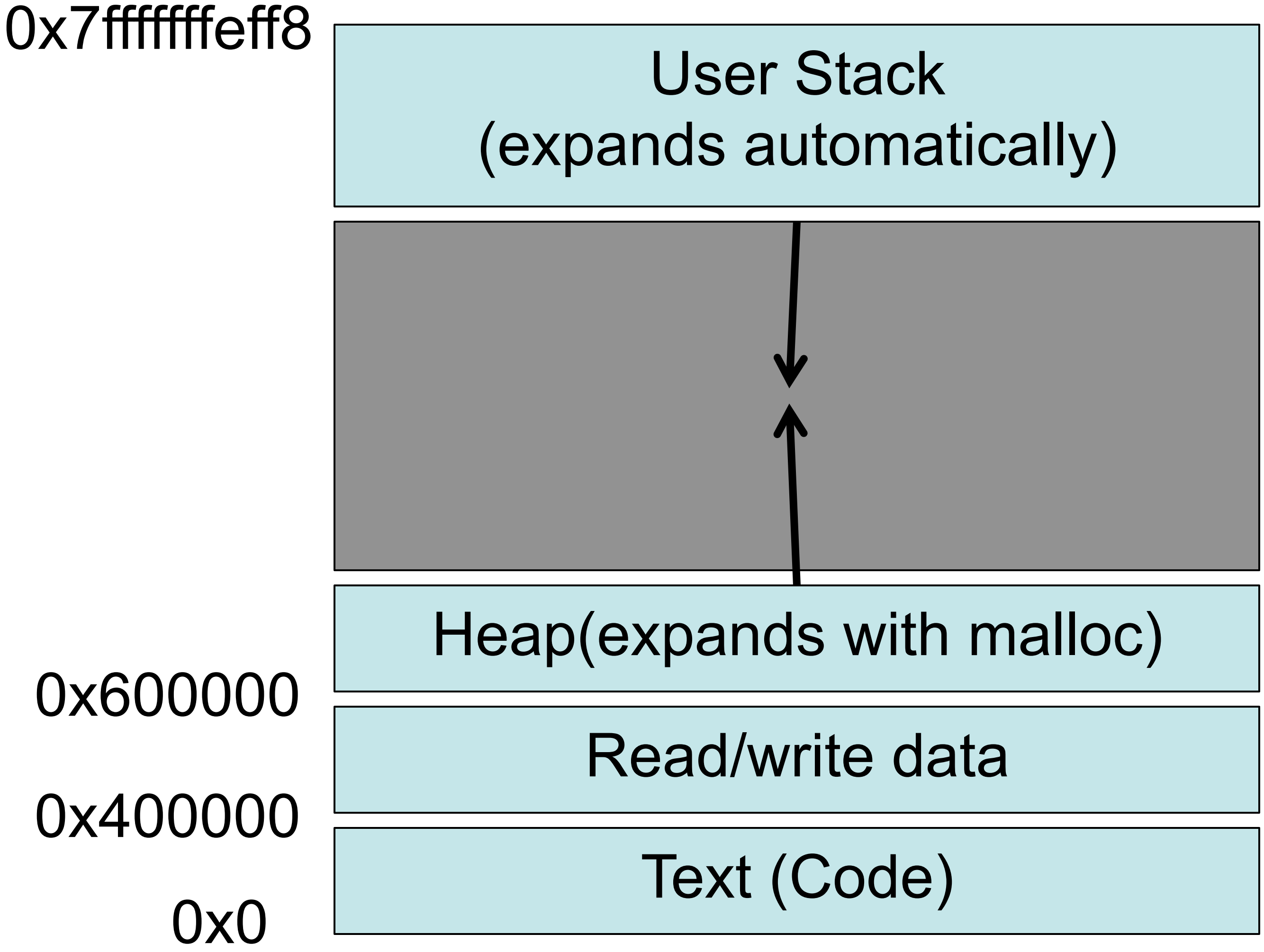
Who can use exploits?

- Those who can access the program
  - Authorized program users
  - Users the program provides a service to
- A key question: user input
  - Is all of it rigorously & completely checked?
  - Typically: *no*
- Some failures: buffer overflows

Buffer overflows

- Many computer languages have arrays
  - Finite (bounded) number of elements
- Some languages (*e.g*., “C”) don’t bounds-check
  - Can alter storage using errant pointers
  - This is called a “buffer overflow”
  - And may be exploitable…

64-bit process virtual memory



0x7fffffffeff8

User Stack
(expands automatically)

Heap(expands with malloc)

0x600000

Read/write data

0x400000

Text (Code)

0x0

Programmed input to a buffer

```
main()
{
  long buf = 0xFEEDDEADBEEFF00D;

#ifdef STACK
  printf("&buf: 0x%lx\n", &buf );
  printf("&get_hex(): 0x%lx\n", &get_hex );
  printf("&problem: 0x%lx\n", &problem );
  printf("&main: 0x%lx\n", &main );
#endif

  get_hex( &buf );
  printf("No problem!\n" );
  exit(0);
}
```

Makes sense to check here

```
void get_hex( long *buf )
{
  long l;
#ifdef STACK
  long *p=&l;
  printf("&l: 0x%lx\n", &l );
  for( l=0; l<16; l++ )
    {
      printf("p: 0x%lx, *p: 0x%lx\n", p, *p );
      ++p;
    }
#endif
  buf = &l;
  while( scanf("%lx", buf ) > 0 )
    {
          buf++;
    }
}
```

This routine is never called

```
void problem()
{
  printf("Problem :-(\n" );
  exit(1);
}
```

Compile w/`–DSTACK` and run

```
spec01:cis551.d$ ./overflow64 </dev/null
&buf: 0x7fffffffe178
&get_hex(): 0x400651
&problem: 0x400634
&main: 0x4006f1
&l: 0x7fffffffe150
p: 0x7fffffffe150, *p: 0x0
p: 0x7fffffffe158, *p: 0x7fffffffe158
p: 0x7fffffffe160, *p: 0x7fffffffe180
p: 0x7fffffffe168, *p: 0x400771
p: 0x7fffffffe170, *p: 0x7fffffffe260
p: 0x7fffffffe178, *p: 0xfeeddeadbeeff00d
p: 0x7fffffffe180, *p: 0x0
p: 0x7fffffffe188, *p: 0x7ffff7aa3a7d
p: 0x7fffffffe190, *p: 0x400550
p: 0x7fffffffe198, *p: 0x7fffffffe268
p: 0x7fffffffe1a0, *p: 0x100000000
p: 0x7fffffffe1a8, *p: 0x4006f1
p: 0x7fffffffe1b0, *p: 0x0
p: 0x7fffffffe1b8, *p: 0xdf03e694e0188f26
p: 0x7fffffffe1c0, *p: 0x400550
p: 0x7fffffffe1c8, *p: 0x7fffffffe260
No problem!
```

An input file for overflow64…

```
spec01:cis551.d$  cat o64_input
0x0
0x7fffffffe158
0x7fffffffe180
0x400634
spec01:cis551.d$
```

Try on new input…

```
spec01:cis551.d$  ./overflow64 <o64_input
&buf: 0x7fffffffe178
&get_hex(): 0x400651
&problem: 0x400634
&main: 0x4006f1
&l: 0x7fffffffe150
p: 0x7fffffffe150, *p: 0x0
p: 0x7fffffffe158, *p: 0x7fffffffe158
p: 0x7fffffffe160, *p: 0x7fffffffe180
p: 0x7fffffffe168, *p: 0x400771
p: 0x7fffffffe170, *p: 0x7fffffffe260
p: 0x7fffffffe178, *p: 0xfeeddeadbeeff00d
p: 0x7fffffffe180, *p: 0x0
p: 0x7fffffffe188, *p: 0x7ffff7aa3a7d
p: 0x7fffffffe190, *p: 0x400550
p: 0x7fffffffe198, *p: 0x7fffffffe268
p: 0x7fffffffe1a0, *p: 0x100000000
p: 0x7fffffffe1a8, *p: 0x4006f1
p: 0x7fffffffe1b0, *p: 0x0
p: 0x7fffffffe1b8, *p: 0xb020326c5790492f
p: 0x7fffffffe1c0, *p: 0x400550
p: 0x7fffffffe1c8, *p: 0x7fffffffe260
Problem :-(
spec01:cis551.d$
```

What have we just done?

- Have entered input from a file
- To alter program flow of control
- In a way that was *impossible*
- Imagine skipping a password check…
- Can you do more?
  - It depends…

Where to learn more:

- http://insecure.org/stf/smashstack.html
- http://paulmakowski.wordpress.com/2011/01/25/smashing-the-stack-in-2011/
- http://nostarch.com/hacking2.htm

Defenses?

- Stack randomization
- No execution of code on stack
- Bounds checking by programmers
- Safe languages:
  - Automatic storage management (G.C.)
  - Strong type checking
  - E.g., Caml, or to a lesser degree, Java

findheap.c

```
#include <stdio.h>
#include <unistd.h>
main()
{
  void *sbrk();
  printf( "0x%lx\n", (long *)
sbrk(0x0) );
}
```

topstack.c

```
#include <stdio.h>
main()
{
  long l, *lp;

  lp = &l;

  while( 1 ){
    printf( "0x%lx: 0x%lx\n", lp, *lp );
    fflush(stdout);
    lp++;
  }

}
```

shinit.c

```
#include <stdio.h>
main()
{
  char *s[2] = {"/bin/sh",
NULL };
  execve( s[0], s, NULL );
}
```

gcc

- *Compilation control command*, not just compiler (cpp, compiler, asm, ld)
- Lots of useful features
- `cc –S prog.c` produces assembly (`prog.s`)
- Also, command line flags to control output file, as in whether to allow execution from stack

shinit.s

```
 .file   "shinit.c"
        .section        .rodata
.LC0:
        .string "/bin/sh"
        .text
.globl main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        movq    %rsp, %rbp
        .cfi_offset 6, -16
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movq    $.LC0, -16(%rbp)
        movq    $0, -8(%rbp)
        movq    -16(%rbp), %rax
        leaq    -16(%rbp), %rcx
        movl    $0, %edx
        movq    %rcx, %rsi
        movq    %rax, %rdi
        call    execve
        leave
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (SUSE Linux) 4.4.1 [gcc-4_4-branch revision 150839]"
        .section        .comment.SUSE.OPTs,"MS",@progbits,1
        .string "ospwg"
        .section        .note.GNU-stack,"",@progbits
```

gdb – GNU debugger

- Very useful tool:
  - Execute program in a controlled environment
  - Examine memory of program
  - Disassemble compiled program
  - “Connect dots” between src and object

Disassembly (gdb shinit)

```
(gdb) disassemble main
Dump of assembler code for function main:
0x0000000000400324 <main+0>:    push   %rbp
0x0000000000400325 <main+1>:    mov    %rsp,%rbp
0x0000000000400328 <main+4>:    sub    $0x10,%rsp
0x000000000040032c <main+8>:    movq   $0x466264,-0x10(%rbp)
0x0000000000400334 <main+16>:   movq   $0x0,-0x8(%rbp)
0x000000000040033c <main+24>:   mov    -0x10(%rbp),%rax
0x0000000000400340 <main+28>:   lea    -0x10(%rbp),%rcx
0x0000000000400344 <main+32>:   mov    $0x0,%edx
0x0000000000400349 <main+37>:   mov    %rcx,%rsi
0x000000000040034c <main+40>:   mov    %rax,%rdi
0x000000000040034f <main+43>:   callq  0x40a190 <__execve>
0x0000000000400354 <main+48>:   leaveq
0x0000000000400355 <main+49>:   retq
End of assembler dump.
(gdb)
```

Examine memory (shinit)

```
(gdb) x/bc 0x466264
0x466264:       47 '/'
(gdb)
0x466265:       98 'b'
(gdb)
0x466266:       105 'i'
(gdb)
0x466267:       110 'n'
(gdb)
0x466268:       47 '/'
(gdb)
0x466269:       115 's'
(gdb)
0x46626a:       104 'h'
(gdb)
0x46626b:       0 '\000'
(gdb)
```

objdump –S shinit

```
0000000000400324 <main>:
  400324:       55                      push   %rbp
  400325:       48 89 e5                mov    %rsp,%rbp
  400328:       48 83 ec 10             sub    $0x10,%rsp
  40032c:       48 c7 45 f0 64 62 46    movq   $0x466264,-0x10(%rbp)
  400333:       00
  400334:       48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
  40033b:       00
  40033c:       48 8b 45 f0             mov    -0x10(%rbp),%rax
  400340:       48 8d 4d f0             lea    -0x10(%rbp),%rcx
  400344:       ba 00 00 00 00          mov    $0x0,%edx
  400349:       48 89 ce                mov    %rcx,%rsi
  40034c:       48 89 c7                mov    %rax,%rdi
  40034f:       e8 3c 9e 00 00          callq  40a190 <__execve>
  400354:       c9                      leaveq
  400355:       c3                      retq
  400356:       90                      nop
  400357:       90                      nop
```