Understanding Common Security Exploits

Sam Hartman

Tom Yu

22 January 2004

Outline of Schedule

Day One: Why security exploits of stack buffer overflows are possible

Day Two: Heap buffer overflows, etc.

Course Resources

Web Page: `http://www.mit.edu/iap/exploits/`

Mailing List: Add yourself to `exploits-students@mit.edu`, or ask us to add you.

Zephyr: Consider subscribing to the `iap-exploits` zephyr class for discussion of the problem sets.

Scope of Course

- This is about understanding, not exploiting.
- We won’t tell you enough to avoid getting caught.
- *Disclaimer: MIT, SIPB, and the instructors neither encourage nor condone the illegal or unethical exercise of any techniques presented here.*

Today's Topics

- Buffer Overflows
- Stacks
- Application Binary Interfaces (ABIs)
- Stack Frames
- Anatomy of a Function Call

Today’s Topics (cont’d)

- i386 ABI Details
- SPARC ABI Details
- Shell Code
- Writing an Exploit
- Useful Tools

Buffer Overflows

Why are Buffer Overflows Possible?

- Nonexistent or incorrect length checking leads to overflows.

- Integer overflows (signed vs unsigned) or failure to understand C arithmetic result in erroneous length checking.

Example of No Length Checking

Many functions such as `strcpy` or `sprintf` will fill a buffer without checking the length. Some functions such as `gets` will even read arbitrary length data from a user.

```
char buffer[12];
gets(buffer);
```

Why Overflows are Harmful

Note that if too many characters are read, the input may change the saved user ID, allowing privilege escalation.

```
char buffer[16];
uid_t saved_uid;
```

Stacks

Why Stacks?

- Computer programs need temporary space for local variables, saved copies of register and where to go when the current task is finished.
- This space needs to be dynamically allocated to allow for recursion.
- A stack fills this role.

A Simple Call Stack

- Basic requirement: store return addresses.
- Procedure call instructions put the address on the stack.
- Return instructions remove the address.
- In practice, functions additionally need arguments and local variables.

Stack Properties

- Like a cafeteria stack of plates.
- Last-In, First-Out (LIFO) structure.
- Top of stack: most recent item added to (“pushed” onto) stack.
- Bottom of stack: oldest item pushed onto stack.
- A register (stack pointer) contains the current position on the stack.

Which Way is Up?

- Most architectures locate “bottom” of stack in “high” memory.
- High addresses: earlier items in stack.
- Low addresses: recent items in stack.
- Stack grows downwards, towards lower addresses.
- Memory diagrams usually draw high addresses at top of page.
- Debuggers usually print lower addresses first.

Application Binary Interfaces (ABIs)

Application Binary Interfaces (ABIs)

- Allow applications to access operating system services, typically via (dynamically loaded) system libraries.
- Explicit specifications for procedure-call conventions, including stack layouts.
- Explicit list of entry points for provided system services.
- Allow compiled application binaries to run on multiple systems providing the same ABI.

Examples We'll Use

- SPARC (CPUs in Sun workstations and servers) running Solaris.
- i386 (a.k.a. x86, Intel386: PC CPUs such as Intel 80386, Pentium, etc.) running Linux.

We'll look at the System V ABI for these CPU architectures.

- Solaris is mostly System V
- Linux ABI calling conventions are like System V.

Stack Frames

i386 Stack Details

- `pushl` pushes a word onto stack.
- `popl` pops a word off the stack.
- `call` pushes return address before jumping to target procedure.
- `%esp` register points to current top of stack (most recently pushed).

The Need for Stack Frames

- Accessing local variables without popping them into a register requires addressing relative to some register pointing into the stack.

- Using stack pointer is problematic: offsets relative to stack pointer change after each push.

- A *frame pointer* register (%ebp on i386) points to top (highest address) of stack frame for the current procedure.

- Locals and arguments addressed relative to frame pointer %ebp.

Anatomy of a Function Call

Example Function Call (C)

```
extern void f(int, int);
void g(void)
{
    f(1, 2);
}
```

Example Function Call (i386 Assembly)

```
g:
; save caller’s frame pointer
        pushl   %ebp
; set up new frame pointer
        movl    %esp, %ebp
; set up local space
        subl    $8, %esp
; push arguments
        pushl   $2
        pushl   $1
        call    f
; "pop" outgoing arguments
        addl    $8, %esp
; restore %ebp
        leave
        ret
```

At Beginning of g( )

Base	Offset	Contents	
		. . .	*High addresses*
%esp	$+0$	return address	
		. . .	*Low addresses*

After Pushing Caller's %ebp

Base	Offset	Contents	
		...	*High addresses*
%esp	$+4$	return address	
%esp	$+0$	caller's %ebp	
		...	*Low addresses*

After Setting Up Local Space

Base	Offset	Contents	
		...	*High addresses*
%ebp	$+4$	return address	
%ebp	$+0$	caller's %ebp	
%ebp	-4	locals...	
%ebp	-8	locals...	$\Leftarrow$%esp
%esp	-4	...	*Low addresses*

After Argument Push

Base	Offset	Contents	
		...	*High addresses*
%ebp	$+4$	return address	
%ebp	$+0$	caller's %ebp	
%ebp	-4	locals...	
%ebp	-8	locals...	
%esp	$+4$	2	
%esp	$+0$	1	
%esp	-4	...	*Low addresses*

Stack Frames and Buffer Overflows

- Stack grows down from the top of memory.
- Locals on the stack grow up.
- Overflows of locals overwrite previously allocated stack space.
- Return address stored on stack. You can overwite the return address to “return” to malicious code.

i386 ABI Details

i386 General-Purpose Register Usage (System V ABI)

Name	Usage	"Owner"
%eax	Return value	callee
%edx	Dividend register (divide operations)	
%ecx	Count register (shift / string operations)	
%ebx	Local register variable	caller
%ebp	Frame pointer	
%esi	Local register variable	
%edi	Local register variable	
%esp	Stack pointer	

i386 Stack Layout (System V C ABI)

Base	Offset	Contents	Frame	
`%ebp`	$4n+8$	argument word n		*High addresses*
		$\ldots$		
`%ebp`	$+8$	argument word 0	Previous	
`%ebp`	$+4$	return address		
`%ebp`	$+0$	caller's `%ebp`		
`%ebp`	-4	x words local space		
		$\ldots$		
`%ebp`	$-4x$	e.g. automatic variables		
`%esp`	$+8$	caller's `%edi`	Current	
`%esp`	$+4$	caller's `%esi`		
`%esp`	$+0$	caller's `%ebx`		*Low addresses*

SPARC ABI Details

SPARC General-Purpose Registers

- SPARC has 32 general-purpose integer registers visible at once.
- `%r0` through `%r7` are global registers `%g0` through `%g7`.
- `%r8` through `%r15` are outgoing registers `%o0` through `%o7`.
- `%r16` through `%r23` are local registers `%l0` through `%l7`.
- `%r24` through `%r31` are incoming registers `%i0` through `%i7`.

Register Windows

- `%r8` through `%r31` are windowed in each procedure.
- Outgoing registers `%o0` through `%o7` of calling procedure are usually incoming registers `%i0` through `%i7` of called procedure.
- Local registers `%l0` through `%l7` are local to each procedure.
- `save` and `restore` instructions shift register windows.
- Procedure call itself does not cause window shift.
- Leaf procedures need not perform `save` and `restore`.

Register Windows (cont'd)

- Finite number of windows.
- Exhaustion triggers spill/fill traps.
- OS responsible for handling window spills/fills by flushing windows to stack.
- Each procedure needs to reserve stack space for window save area.

Register Windows Illustrated

Caller

%i7 (%r31) ... %i0 (%r24)	ins
%l7 (%r23) ... %l0 (%r16)	locals
%o7 (%r15) ... %o0 (%r8)	outs

Callee

%i7 (%r31) ... %i0 (%r24)	ins
%l7 (%r23) ... %l0 (%r16)	locals
%o7 (%r15) ... %o0 (%r8)	outs

Uses of Specific Registers

- `%g0` always reads zero, and writes to it are ignored.
- The `call` instruction stores its own address into `%o7`.
- Due to windowing, `%i7` contains address of caller's `call` instruction.

SPARC System V ABI Register Usage

- `%o6` and `%i6` are `%sp` (stack pointer) and `%fp` (frame pointer).
- `%sp` must point to a 16-word window save area.
- `%l0`,. . . , `%l7`, `%i0`,. . . , `%i7` written to window save area by system during a spill trap; restored during fill trap.
- Windowing causes caller's `%sp` to be the callee's `%fp`.
- `%i0` is the return value (`%i0` becomes the caller's `%o0`).
- `%g5` through `%g7` reserved for the system.

SPARC Stack Frame (System V C ABI)

Base	Offset	Contents	Frame
`%fp`	$+92$	callee's arguments 6, ...	*High addresses*
		argument dump	
`%fp`	$+68$	for callee's `%i0`–`%i5`	
`%fp`	$+64$	`struct` / `union` return pointer	Previous (caller)
`%fp`	$+60$	spilled `%i7` (return address -8)	
`%fp`	$+56$	spilled `%i6` (`%fp`)	
		spilled `%l1`,..., `%l7`, `%i0`,..., `%i5`	
`%fp`	$+0$	spilled `%l0`	
`%fp`	-4	y words local space	
		...	
`%fp`	$-4y$	e.g. automatic variables	
`%sp`	$+88+4x$	x words compiler scratch space	
		...	Current
`%sp`	$+92$	outgoing arguments 6, ...	(callee)
`%sp`	$+68$	outgoing arguments 0–5	
`%sp`	$+64$	`struct` / `union` return pointer	*Low*
`%sp`	$+0$	16-word window save area	*addresses*

Register Window Complications for Exploits

- Return address (in window save area) is lower in memory than locals.
- Even then, only written to stack during window spills.
- To exploit a procedure, overwrite *caller's* return address by overflowing locals into caller's window save area.
- Even then, fails if caller's register window not flushed yet.

Shell Code

Shell Code

- Compact machine code you can stick into a buffer.

- Called “shell code” because traditionally, when executed, starts a new Unix command shell.

Shell Code Considerations

- Needs to be small to fit in buffer without crashing the application.
- Needs to be location independent.
- Should be properly aligned.

Landing Pads

- Exact location of start of shell code possibly not known.
- Landing pad allows execution to safely start anywhere within a range of addresses.
- Use “no operation” (NOOP) opcodes or short relative jumps in landing pad.

Location Independent Code

- Make syscalls directly rather than using library functions. Calling library functions requires access to the linker.
- Use addresses relative to instruction pointer or stack pointer.
- Avoid any relocations for data references.

Sample Location Independent code

This code points `%eax` at the string `foo`. (It then proceeds to crash.)

```
        call mark
mark:   pop %eax
        addl $(foo-mark), %eax
foo:    .string "foo"
```

Writing Direct Syscalls

- Write a simple C program that calls the syscall you want to make.
- Compile the program and link statically against the C library.
- Step through the debugger looking at generated assembly.
- Understand what the registers and stack are when the code traps into the kernel.

Advanced Shell Code Considerations

- You may need to avoid using certain characters such as control characters or certain characters special to the protocol you attack.

- Some shells such as Solaris `/bin/sh` require that all uids be the same. If exploiting a set-uid program you may need to call `setuid(0)`.

Writing an Exploit

Exploiting a Buffer Overflow

- Insert shell code somewhere and point the return address so that control flow intersects your shell code.
- If buffer large enough, can cause the shell code to end up in buffer and just overwrite the return address.
- Otherwise, may be able to put the shell code higher on the stack than the buffer.

Getting Shell Code in the Buffer

- Interact with the program enough to get shell code into buffer.
- May involve encoding shell code in some network protocol.
- May involve participating in protocol up to the point where buffer will be read into.
- Common encodings: URL escaping and MIME.

Finding the New Return Address

- Start by running the program in the debugger and finding the address of the buffer. Adjust depending on where your shell code is placed.

- Note that the top of the stack may change somewhat between runs of the program.

- If you don't have access to run the program in a debugger, you can guess and work down from top of stack.

Useful Tools

Displaying Instruction in GDB

- `(gdb) disp/i $pc`

- Then every time GDB stops you find the current instruction:

```
Breakpoint 1, 0x10d34 in main ()
1: x/i $pc  0x10d34 <main+8>: add  %g2, 0x224, %o1
```

- Use `si` to move forward one instruction.

Getting Assembly From Compiler

```
gcc -S file.c
```

- Look at `file.s` for assembly language output.
- Optimization settings significantly influence compiler output.
- With `gcc`, sometimes the output when using `gcc -O` is more readable than unoptimized output.

Using Objdump to Disassemble

```
objdump -j .text -d overflow.o
00000014 <main>:
  14:   55                              push   %ebp
  15:   89 e5                           mov    %esp,%ebp
  17:   83 ec 08                        sub    $0x8,%esp
  1a:   e8 fc ff ff ff                  call   1b <main+0x7>
  1f:   c9                              leave
  20:   c3                              ret
```

Extracting Binary with Objcopy

- Once you have shell code, you can use `objcopy` to extract the processor instructions from the object file.
- `objcopy -j .text -O binary` *infile*`.o` *outfile*`.bin`
- Test using `objdump -D -b binary -m` *architecture* *outfile*`.bin`
- May also have to give endianness flag (`-EL` or `-EB`) to objdump.

Resources

CPU Architecture References

- SPARC International, Inc., *The SPARC Architecture Manual, Version 9*, Prentice-Hall, Inc., 2000. Downloadable from `http://www.sparc.org/`

- Intel Corporation, *Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual*, Intel, 1996.

System V ABI References

System V ABI documentation may be obtained from The Santa Cruz Operation, Inc., `http://www.sco.com/`

- The Santa Cruz Operation, Inc., *System V Application Binary Interface, Intel386 Architecture Processor Supplement, Fourth Edition*, SCO, 1996.

- The Santa Cruz Operation, Inc., *System V Application Binary Interface, SPARC Processor Supplement, Third Edition*, SCO, 1996.

Additional Resources

Intel Corporation: `http://www.intel.com/`

Sparc International, Inc.: `http://www.sparc.org/`

Bugtraq: `http://www.securityfocus.com/`

Phrack: `http://www.phrack.org/`

SIPB's documentation archive:

`http://www.mit.edu/afs/sipb.mit.edu/contrib/doc/`

in particular, look at `specs/hardware/ic/cpu/`

and `specs/software/sysv-abi/`

Additional Resources (cont'd)

These slides are available at

`http://www.mit.edu/iap/exploits/exploits01.pdf`

Problem Set 1

`http://www.mit.edu/iap/exploits/ps1.pdf`

Course Home Page:

`http://www.mit.edu/iap/exploits/`