CIS433/533 - Computer and Network Security
Software Security

Professor Kevin Butler
Winter 2010
Buffer Overflow

- Very common attack mechanism
  - from 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- prevention techniques known
- still of major concern due to
  - legacy of widely deployed buggy
  - continued careless programming techniques
Buffer Overflow Basics

- caused by programming error
- allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker
Buffer Overflow Example

```c
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
Example

```c
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];

    next_tag(str1);
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(\%s), str2(\%s), valid(\%d)\n", str1, str2, valid);
}
```

$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
Buffer Overflow Attacks

- to exploit a buffer overflow an attacker:
  - must identify a buffer overflow vulnerability in some program
    - inspection, tracing execution, fuzzing tools
  - understand how buffer is stored in memory and determine potential for corruption
Programming Language History

- At machine level all data an array of bytes
  - Interpretation depends on instructions used
- Modern high-level languages (e.g., Java, Python) have a strong notion of type and valid operations
  - Not vulnerable to buffer overflows
  - Does incur overhead, some limits on use
- C and related languages have high-level control structures, but allow direct access to memory
- Best of both worlds?
  - Vulnerable to buffer overflow
  - Have a large legacy of widely used, unsafe, and hence vulnerable code
Function Calls & Stack Frames

- Return Addr
- Old Frame Pointer
- param 2
- param 1

Q:
- Return Addr in P
- Old Frame Pointer
- local 1
- local 2

P:
- Frame Pointer
- Stack Pointer
Stack Buffer Overflow

- occurs when buffer is located on stack
  - used by Morris Worm (used unsafe `gets` in `fingerd`)
  - Aleph One paper popularized it

- have local variables below saved frame pointer and return address
  - hence overflow of a local buffer can potentially overwrite these key control items

- attacker overwrites return address with address of desired code
  - program, system library or loaded in buffer
Programs and Processes

- Process image in main memory
- Kernel Code and Data
- Stack
- Spare Memory
- Heap
- Program File
- Global Data
- Program Machine Code
- Global Data
- Program Machine Code
- Process Control Block
- Top of Memory
- Bottom of Memory
In Practice

- How it works
void getinp(char *inp, int siz)
{
    puts("Input value: ");
    fgets(inp, siz, stdin);
    printf("buffer3 getinp read %s
", inp);
}

void display(char *val)
{
    char tmp[16];
    sprintf(tmp, "read val: %s\n", val);
    puts(tmp);
}

int main(int argc, char *argv[])
{
    char buf[16];
    getinp(buf, sizeof(buf));
    display(buf);
    printf("buffer3 done\n");
}

What’s wrong with this code?
Another Stack Overflow

```
$ cc -o buffer3 buffer3.c

$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
```
Shellcode

- code supplied by attacker
  - often saved in buffer being overflowed
  - traditionally transferred control to a shell
- machine code
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently have automated sites/tools
Shellcode Development

• illustrate with classic Intel Linux shellcode to run Bourne shell interpreter

• shellcode must
  ‣ marshall argument for execve() and call it
  ‣ include all code to invoke system function
  ‣ be position-independent
  ‣ not contain NULLs (C string terminator)
Example Shellcode

**NOP sled**

```assembly
nop               // end of nop sled
nop
jmp find         // jump to end of code
cont: pop %esi   // pop address of sh off stack into %esi
xor %eax,%eax   // zero contents of EAX
mov %al,0x7(%esi) // copy zero byte to end of string sh (%esi)
lea (%esi),%ebx  // load address of sh (%esi) into %ebx
mov %ebx,0x8(%esi) // save address of sh in args[0] (%esi+8)
mov %eax,0xc(%esi) // copy zero to args[1] (%esi+c)
mov $0xb,%al    // copy execve syscall number (11) to AL
mov %esi,%ebx    // copy address of sh (%esi) to %ebx
lea 0x8(%esi),%ecx // copy address of args (%esi+8) to %ecx
lea 0xc(%esi),%edx // copy address of args[1] (%esi+c) to %edx
int $0x80        // software interrupt to execute syscall
find: call cont  // call cont which saves next address on stack
sh: .string "/bin/sh " // string constant
args: .long 0     // space used for args array
       .long 0     // args[1] and also NULL for env array
```

**Assembled x86 code**

Machine code (alphanumeric representation)

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 20 20 20 20 20 20 20 20
```
Stack Overflow Variants

- target program can be:
  - a trusted system utility
  - network service daemon
  - commonly used library code, e.g. image

- shellcode functions
  - spawn shell
  - create listener to launch shell on connect
  - create reverse connection to attacker
  - flush firewall rules
  - break out of chroot environment
Buffer Overflow Defenses

- buffer overflows are widely exploited
- large amount of vulnerable code in use
  - despite cause and countermeasures known
- two broad defense approaches
  - compile-time - harden new programs
  - run-time - handle attacks on existing programs
Compile-Time Defenses

- use a modern high-level languages with strong typing
  - not vulnerable to buffer overflow
  - compiler enforces range checks and permissible operations on variables
- do have cost in resource use
- and restrictions on access to hardware
  - so still need some code in C-like languages
Compile-Time Defenses

- proposals for safety extensions to C (e.g., CCured, Cyclone)
  - performance penalties
  - must compile programs with special compiler
- have several safer standard library variants
  - new functions, e.g. strlcpy()
  - safer re-implementation of standard functions as a dynamic library, e.g. Libsafe
Stack Protection

- **StackGuard**
  - Push a ‘canary’ on the stack between the local vars and the return pointer
  - Overwrite of canary indicates a buffer overflow
  - Requires changes to the compiler

- **Q:** Would this solve the problem?

- **Thorough summary:**
Non-Executable Address Space

- Runtime defense
- use virtual memory support to make some regions of memory non-executable
  - e.g. stack, heap, global data
  - need h/w support in MMU
  - long existed on SPARC / Solaris systems
  - recent on x86 Linux/Unix/Windows systems
    - no-execute bit in MMU
- issues: support for executable stack code
  - need special provisions
Address Space Randomization

- manipulate location of key data structures
  - stack, heap, global data
  - using random shift for each process
  - have large address range on modern systems means wasting some has negligible impact
- also randomize location of heap buffers
- and location of standard library functions
- Solves all of our problems?
Guard Pages

- place guard pages between critical regions of memory
  - flagged in MMU as illegal addresses
  - any access aborts process
- can even place between stack frames and heap buffers
  - at execution time and space cost
Other Overflow Attacks

- Wide range of other attack variants
  - stack overflow variants
  - heap overflow
  - global data overflow
  - format string overflow
  - integer overflow

- more likely to be discovered in future

- some cannot be prevented except by coding to prevent originally
Replacement Stack Frame

- stack overflow variant just rewrites buffer and saved frame pointer
  - so return occurs but to dummy frame
  - return of calling function controlled by attacker
  - used when have limited buffer overflow
  - e.g. off by one

- limitations
  - must know exact address of buffer
  - calling function executes with dummy frame
Return to libc

- stack overflow variant replaces return address with standard library function
  - response to non-executable stack defences
  - attacker constructs suitable parameters on stack above return address
  - function returns and library function executes
    - e.g. `system("shell commands")`
  - attacker may need exact buffer address
  - can even chain two library calls
- Return-oriented rootkits (Shacham et al., UCSD)
Heap Overflow

- also attack buffer located in heap
  - typically located above program code
  - memory requested by programs to use in dynamic data structures, e.g. linked lists

- no return address
  - hence no easy transfer of control
  - may have function pointers can exploit
  - or manipulate management data structures

- defenses: non executable or random heap
Java World

- Type Safe Language
  - No buffer/heap/ptr overflows
  - No unsafe casts
  - Still have integer overflows?

- Java Virtual Machine
  - Interpret bytecode (or compile together)
  - Security Manager (reference monitor for JVM)

- Q: What is the trust model of a Java application?
• From C to Memory-safe C Translator
  ‣ Find the minimum number of runtime checks to ensure memory safety

• Classify Pointers
  ‣ Safe
  ‣ Wild
    ▪ Need runtime checks for wild pointers

• Runtime Checks
  ‣ Similar to declassifiers in DLM
  ‣ Written by hand, in general
C Analysis

- Assume Type Safety in Analysis
  - On what basis?
  - Trust that the programmer does not subvert

- Is this a reasonable assumption?
  - Unsound analysis
    - False negatives are possible
  - Sound analysis
    - If no unsafe behavior relative to analysis can be assumed
    - False positives are possible

- Actually, lots of work in this area
- Used in production code: Microsoft
Source Code Analysis

- Shallow tools for bug finding
  - Prefix, Prefast -- Microsoft

- Companies that will check your code
  - Coverity -- based on MC

- Deep tools for verifying correctness
  - SLAM -- for device drivers

- Add security to legacy code
  - Generate LSM
  - Generate reference monitor for X Server

- Lots of other topics
  - Privilege separation, domain transition, error reporting
Driver Verification

void LeakSample(BOOLEAN Option1)
{
    NTSTATUS Status;
    KIRQL OldIrql;
    BufInfo *pBufInfo;
    KeAcquireSpinLock(MyLock,&OldIrql);
    //...
    if (Option1) {
        pBufInfo = ExAllocatePoolWithTag(NonPagedPool,
                                         sizeof(BufInfo), 'fuB_');
        if (NULL==pBufInfo) {
            return STATUS_NO_MEMORY;
        }
        //...
        KeReleaseSpinLock(MyLock, OldIrql);
        return STATUS_SUCCESS;
    }
    //...
}

- Memory leak of spin lock resource

PREfast catches this
void LeakSample(BOOLEAN Option1)
{
    NTSTATUS Status;
    KIRQL OldIrql;
    BufInfo *pBufInfo;
    KeAcquireSpinLock(MyLock,&OldIrql);
    //...
    if (Option1) {
        pBufInfo = ExAllocatePoolWithTag(NonPagedPool,
            sizeof(BufInfo), 'fuB_');
        if (NULL==pBufInfo) {
            KeReleaseSpinLock(MyLock, OldIrql);
            return STATUS_NO_MEMORY;
        }
        KeReleaseSpinLock(MyLock, OldIrql);
        return STATUS_SUCCESS;
    }
    //...
    KeReleaseSpinLock(MyLock, OldIrql);
    return STATUS_SUCCESS;
}
//...

- Make sure lock is released (match Acquires with Releases)
Security Typed Languages

- **Key:**
  - tag data & monitor flows
  - e.g., language: Jif

- **RM's tag actual data**
  - all data/processes have label
  - central security monitor checks operations, data access against policy

- **Security-typed languages use virtual tags**
  - data types are labeled
  - type checker validates flows

Label all data
Monitor flows
Build on type safety

- A type-safe language maintains the semantics of types. E.g. can’t add int’s to Object’s.
- Type-safety is compositional. A function promises to maintain type safety.

**Example 1**
Object obj;
int i;
obj = obj + i;

**Example 2**
String proc_obj(Object o);
...
main()
{
   Object obj;
   String s = proc_obj(obj);
   ...
}
Labeling types

Example 1
int{high} h1, h2;
int{low} l;
l = 5;
h2 = l;
h1 = h2 + 10;
l = h2 + l;

- Key insight:
  - label types with security levels
- Security-typing is compositional

Example 2
String{low} proc_obj(Object{high} o);
...
main()
{
    Object{high} obj;
    String{low} s;
    s = proc_obj(obj);
    ...
}
public class SecretMessages[principal alice, principal bob]
{
    String{alice:} aliceInstructions;
    String{bob:} bobInstructions;

    public SecretMessages(String{alice:} ai, String{bob:} bi) {
        aliceInstructions = ai;
        bobInstructions = bi;
    }
    ...

    public String{bob:} leak() {
        bobInstructions = aliceInstructions;
        return bobInstructions;
    }
}
public class SecretMessages[label alice, label bob] {
    String{*alice} aliceInstructions;
    String{*bob} bobInstructions;

    public SecretMessages(String{*alice} ai, String{*bob} bi) {
        aliceInstructions = ai;
        bobInstructions = bi;
    }

    ...}

    public String{*bob} implicitLeak() {
        try {
            if (aliceInstructions.equals("Attack at dawn"))
                bobInstructions = "Attack at dawn";
        } catch (NullPointerException e) {}
        return bobInstructions;
    }
}
Declassification

- MLS is too restrictive
- Examples:
  - Encryption
  - Distributed auction
  - Password check
- Solutions:
  - Declassification
    - Reduce the level of data -- tolerable leakage