CIS 433/533 - Computer and Network Security
Software Security
Prof. Kevin Butler
Winter 2013
Programming

• Why do we write programs?
  ‣ Function

• What functions do we enable via our programs?
  ‣ Some we want -- some we don’t need
  ‣ Adversaries take advantage of such “hidden” function
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);

What if packet is larger than 1000 bytes?
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);

What if packet is only 1004 bytes?
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");
}
$ cc -o buffer3 buffer3.c

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

$ ./buffer3
Input value:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXXXXXXXXX

buffer3 done
Segmentation fault (core dumped)
Address Space Layout

- Write beyond variable limit
  - Can write the without limits in some languages
- Can impact values
  - In heap, on stack, in data
- Can impact execution integrity
  - Can jump to arbitrary points in the program
    - Function pointers
    - Return addresses
Buffer Overflow

• How it works

Stack Frame

Previous Function

Func Parameters

Return Address

Local Var

Buffer

Local Var

Evil Code

New Rtn
Overflow of Local Variables

• Don’t need to modify return address
  ‣ Local variables may affect control

• What kinds of local variables would impact control?
  ‣ Ones used in conditionals (example)
  ‣ Function pointers

• What can you do to prevent that?
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);

What if we allocate the packet buffer on the heap?
Heap Overflow

- Overflows may occur on the heap also
  - Heap has data regions and metadata

- Attack
  - Write over heap with target address (heap spraying)
  - Hope that victim uses an overwritten function pointer before program crashes
Another Simple Program

```c
int size = BASE_SIZE;
char *packet = (char *)malloc(1000);
char *buf = (char *)malloc(1000+BASE_SIZE);

strcpy(buf, FILE_PREFIX);

size += PacketRead(packet);

if (size < sizeof(buf)) {
    strcat(buf, packet);
    fd = open(buf);
}
```

Any problem with this conditional check?
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessQuery("Select", partof(packet));
Integer Overflow

• Signed variables represent positive and negative values
  ‣ Consider an 8-bit integer: -128 to 127
  ‣ Weird math: \(127+1 = ???\)

• This results in some strange behaviors
  ‣ \texttt{size += PacketRead(packet)}
    • What is the possible value of \texttt{size}?
  ‣ \texttt{if (size < sizeof(buf))} {
    • What is the possible result of this condition?

• How do we prevent these errors?
Character Strings

- String formats
  - Unicode
    - ASCII -- 0x00 -- 0x7F
    - non-ASCII -- 0x80 -- 0xF7
    - Also, multi-byte formats
  - Decoding is a challenge
    - URL: [IPaddr]/scripts/..%c0%af../winnt/system32
    - Decodes to /winnt/system32
  - Markus Kuhn’s page on Unicode resources for Linux
    - www.cl.cam.ac.uk/~mgk25/unicode.html
HI, THIS IS YOUR SON’S SCHOOL. WE’RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR – DID HE BREAK SOMETHING? IN A WAY–

DID YOU REALLY NAME YOUR SON Robert’); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE’VE LOST THIS YEAR’S STUDENT RECORDS. I HOPE YOU’RE HAPPY.

AND I HOPE YOU’VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Parsing Errors

• Have to be sure that user input can only be used for expected function
  ‣ **SQL injection**: user provides a substring for an SQL query that changes the query entirely (e.g., add SQL operations to query processing)

SELECT fieldlist
FROM table
WHERE field = 'anything' OR 'x'='x';

• Goal: format all user input into expected types and ranges of values
  ‣ Integers within range
  ‣ Strings with expected punctuation, range of values

• Many scripting languages convert data between types automatically -- are not **type-safe** -- so must be extra careful
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

- 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

Assembled x86 code

<table>
<thead>
<tr>
<th>Address Code</th>
<th>Machine code (alphanumeric representation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89</td>
<td>90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89</td>
</tr>
<tr>
<td>46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1</td>
<td>46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1</td>
</tr>
<tr>
<td>ff ff ff 2f 62 69 6e 20 20 20 20 20 20</td>
<td>ff ff ff 2f 62 69 6e 20 20 20 20 20 20</td>
</tr>
</tbody>
</table>
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 choot 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
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

• Thorough summary:
Buffer Overflow Defense

<table>
<thead>
<tr>
<th>Previous Function</th>
<th>Func Parameters</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CANARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Var</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local Var</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **“Canary” on the stack**
  - Random value placed between the local vars and the return address
  - If canary is modified, program is stopped

- Are we done?
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
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
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