Language Security

Lecture 40
(from notes by G. Necula)

Lecture Outline

• Beyond compilers
  - Looking at other issues in programming language design and tools

• C
  - Arrays
  - Exploiting buffer overruns
  - Detecting buffer overruns

Platitudes

• Language design has influence on
  - Efficiency
  - Safety
  - Security

C Design Principles

• Small language
• Maximum efficiency
• Safety less important

• Designed for the world as it was in 1972
  - Weak machines
  - Superhuman programmers (or so they thought)
  - Trusted networks

Arrays in C

char buffer[100];
Declares and allocates an array of 100 chars

C Array Operations

char buf1[100], buf2[100];

Write:
buf1[0] = 'a';

Read:
return buf2[0];
What’s Wrong with this Picture?

```c
int i;
for(i = 0; buf1[i] != '\0'; i++) {
    buf2[i] = buf1[i];
}
buf2[i] = '\0';
```

Indexing Out of Bounds

The following are all well-typed C and may generate no run-time errors

```c
char buffer[100];
buffer[-1] = 'a';
buffer[100] = 'a';
buffer[100000] = 'a';
```

Why?

- Why does C allow out-of-bounds array references?
  - Proving at compile-time that all array references are in bounds is impossible in most languages
  - Checking at run-time that all array references are in bounds is "expensive"
    - But it is even more expensive to skip the checks

Code Generation for Arrays

- The C code:
  ```c
  buf1[i] = 1; /* buf1 has type int[] */
  ```
  ```asm
  r1 = &buf1;
  r2 = load i;
  r3 = r2 * 4;
  if r3 < 0 then error;
  r5 = load limit of buf1;
  if r3 >= r5 then error;
  r4 = r1 + r3
  store r4, 1
  ```
  - Finding the array limits is non-trivial
  - Costly!

C vs. Java

- C array reference typical case
  - Offset calculation
  - Memory operation (load or store)
- Java array reference typical case
  - Offset calculation
  - Memory operation (load or store)
  - Array bounds check
  - Type compatibility check (for some arrays)

Buffer Overruns

- A buffer overrun writes past the end of an array
- *Buffer* usually refers to a C array of char
  - But can be any array
- So who’s afraid of a buffer overrun?
  - Can cause a core dump
  - Can damage data structures
  - What else?
Stack Smashing

Buffer overruns can alter the control flow of your program!

char buffer[100]; /* stack allocated array */

An Overrun Vulnerability

void foo(char in[]) {
    int i = 0;
    for(i = 0; in[i] != '\0'; i++)
        { buffer[i] = in[i]; }
    buffer[i] = '\0';
}

An Interesting Idea

char in[104] = {'...','...','magic 4 chars'}
foo(in): (**)

Discussion

• So we can make foo jump wherever we like.

• How is this possible?

• Unanticipated interaction of two features:
  - Unchecked array operations
  - Stack-allocated arrays and return addresses
  - Knowledge of frame layout allows prediction of where
    array and return address are stored
  - Note the "magic cast" from char's to an address

The Rest of the Story

• Say that foo is part of a network server and the in originates in a received message
  - Some remote user can make foo jump anywhere!

• But where is a "useful" place to jump?
  - Idea: Jump to some code that gives you control of the host system (e.g. code that spawns a shell)

• But where to put such code?
  - Idea: Put the code in the same buffer and jump there!

The Plan

• We'll make the code jump to the following code:
  - In C: exec("/bin/sh");
  - In assembly (pretend):
    mov $a0, 15 ; load the syscall code for "exec"
    mov $a1, &Ldata ; load the command
    syscall ; make the system call
    Ldata: .byte '/','b','i','n','/','s','h',0 ; null-terminated
  - In machine code: 0x20, 0x42, 0x00, ...
The Plan

char in[104] = { 104 magic chars }
foo(in);

• The last 4 bytes in "in" must equal the start of buffer
• Its position might depend on many factors!

Guess the Location of the Injected Code

• Trial & error: gives you a ballpark
• Then pad the injected code with NOP
  - E.g. add $0, $1, 0x2020
  - stores result in $0 which is hardwired to 0 anyway
  - Encoded as 0x20202020

• Works even with an approximate address of buffer!

More Problems

• We do not know exactly where the return address is
  - Depends on how the compiler chose to allocate variables in
    the stack frame
• Solution: pad the buffer at the end with many copies
  of the "magic return address X"

Even More Problems

• The most common way to copy the bad code in a
  stack buffer is using string functions: strcpy,
  strcat, etc.
• This means that buf cannot contain 0x00 bytes
  - Why?
• Solution:
  - Rewrite the code carefully
  - Instead of "addiu $4,$0,0x0015 (code 0x20400015)
  - Use "addiu $4,$0,0x1126; subiu $4, $4, 0x1111"

The State of C Programming

• Buffer overruns are common
  - Programmers must do their own bounds checking
  - Easy to forget or be off-by-one or more
  - Program still appears to work correctly
• In C w.r.t. to buffer overruns
  - Easy to do the wrong thing
  - Hard to do the right thing

The State of Hacking

• Buffer overruns are the attack of choice
  - 40-50% of new vulnerabilities are buffer overrun
    exploits
  - Many recent attacks of this flavor: Code Red,
    Nimda, MS-SQL server
• Highly automated toolkits available to exploit
  known buffer overruns
  - Search for "buffer overruns" yields > 25,000 hits
The Sad Reality

- Even well-known buffer overruns are still widely exploited
  - Hard to get people to upgrade millions of vulnerable machines
- We assume that there are many more unknown buffer overrun vulnerabilities
  - At least unknown to the good guys

Static Analysis to Detect Buffer Overruns

- Detecting buffer overruns before distributing code would be better
- Idea: Build a tool similar to a type checker to detect buffer overruns
- Joint work by Alex Aiken, David Wagner, Jeff Foster, at Berkeley

Focus on Strings

- Most important buffer overrun exploits are through string buffers
  - Reading an untrusted string from the network, keyboard, etc.
- Focus the tool only on arrays of characters

Idea 1: Strings as an Abstract Data Type

- A problem: Pointer operations & array dereferences are very difficult to analyze statically
  - Where does *a point?
  - What does buf[j] refer to?
- Idea: Model effect of string library functions directly
  - Hard code effect of strcpy, strcat, etc.

Idea 2: The Abstraction

- Model buffers as pairs of integer ranges
  - Alloc min allocated size of the buffer in bytes
  - Length max number of bytes actually in use
- Use integer ranges \([x,y] = \{x, x+1, \ldots, y-1, y\}\)
  - Alloc & length cannot be computed exactly

The Strategy

- For each program expression, write constraints capturing the alloc and len of its string subexpressions
- Solve the constraints for the entire program
- Check for each string variable \(s\)
  \[\text{len}(s) \leq \text{alloc}(s)\]
The Constraints

- `char s[n];`  \( n \leq \text{alloc}(s) \)
- `strcpy(dst, src)`  \( \text{len}(src) \leq \text{len}(dst) \)
- `p = strdup(s)`  \( \text{len}(s) \leq \text{len}(p) \)  \&  \( \text{len}(s) \leq \text{alloc}(p) \)
- `p[n] = '\0'`  \( n+1 \leq \text{len}(p) \)

Constraint Solving

- Solving the constraints is akin to solving dataflow equations (e.g., constant propagation)
- Build a graph
  - Nodes are \( \text{len}(s), \text{alloc}(s) \)
  - Edges are constraints \( \text{len}(s) = \text{len}(t) \)
- Propagate information forward through the graph
  - Special handling of loops in the graph

Using Solutions

- Once you've solved constraints to extract as much information as possible, look to see if \( \text{len}(s) = \text{alloc}(s) \)
  - is necessarily true. If not, may have a problem.
- For example, if `b` is parameter about which we know nothing, then in `char s[100]; strcpy(s, b);` \( \text{len}(s) = \text{alloc}(s) \) will not simplify to True.

Results

- Found new buffer overruns in `sendmail`
- Found new exploitable overruns in Linux `nettools` package
- Both widely used, previously hand-audited packages

Limitations

- Tool produces many false positives
  - 1 out of 10 warnings is a real bug
- Tool has false negatives
  - Unsound---may miss some overruns
- But still productive to use

Summary

- Programming language knowledge useful beyond compilers
- Useful for programmers
  - Understand what you are doing!
- Useful for tools other than compilers
  - Big research direction