Language Security

Lecture 19

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
  - Safety
  - Efficiency
  - Security

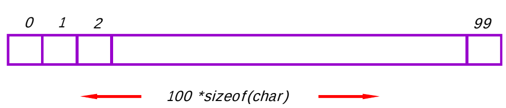
C Design Principles

- Small language
- Maximum efficiency
- Safety less important
- Designed for the world in 1972
  - Weak machines
  - 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?

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


Indexing Out of Bounds

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

```
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 very difficult (impossible in C)
  - Checking at run-time that all array references are in bounds is expensive

Code Generation for Arrays

```
buf1[i] = 1;    /* buf1 has type int[] */

r1 = load &buf1;
r2 = load i;
r3 = r2 * 4;
r4 = r1 + r3
store r4, 1
```


Discussion

- 5 instructions worst case
- Often &buf1 and i already in registers
  - Saves 2 instructions
- Many machines have indirect loads/stores
  - store r1[r3], 1
  - Saves 1 instruction
- Best case 2 instructions
  - Offset calculation and memory operation

Code Generation for Arrays with Bounds Checks

```
buf1[i] = 1;    /* buf1 has type int[] */

r1 = load &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
```

Discussion

- Lower bounds check can often be removed
  - Easy to prove statically that index is positive
- Upper bounds check hard to remove
  - Leaves a conditional in instruction stream
- In C, array limits not stored with array
  - Knowing the array limit for a given reference is non-trivial

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 stores)

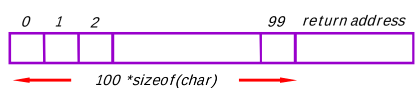
Buffer Overruns

- 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 damage data structures
  - Cause a core dump
  - What else?

Stack Smashing

Buffer overruns can alter the control flow of your program!

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



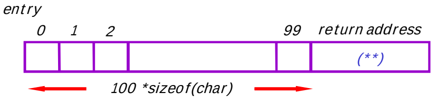
An Overrun Vulnerability

```
void foo(char buf1[]) {
   char buf2[100];
   int i = 0;
   for(i = 0; buf1[i] != '\0'; i++)
       { buf2[i] = buf1[i]; }
   buf2[i] = '\0';
}
```

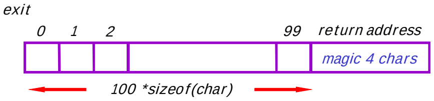

An Interesting Idea

```
char buf[104] = { ' ',…,' ', magic 4 chars }
foo(buf);   (**)
```

Foo entry



Foo exit



Discussion

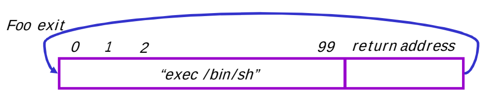
- So we can make foo jump wherever we like.
- How is this possible?
- Unanticipated interaction of two features:
  - Unchecked array operations
  - Stack-allocated arrays
    - Knowledge of frame layout allows prediction of where array and return address are stored
  - Note the "magic cast" from char's to an address

The Rest of the Story

- We can make foo jump anywhere.
- But where is a useful place to jump?
- Idea: Put our own code in the buffer and jump there!

The Plan

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



Details

- "exec /bin/sh"
  - Easy to write in assembly code
  - Make all jumps relative
- Be careful not to have null's in the code (why?)

More Details

- Overwrite return address with start of buffer
  - Harder
  - Need to guess where buffer in called routine starts (trial & error)
  - Pad front of buffer with NOPs
    - Guess need not be exact; just land somewhere in NOPs

And More Details

- Overwrite return address
  - Don't need to know exactly where return address is
  - Just pad end of buffer with multiple copies of new return address X

```
char buf[104] =
  "NOPS … /bin/exec sh XXXXXXXXXX"
foo(buf);
```

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 wrt 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
- 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 with David Wagner & Jeff Foster

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
  - *Size* allocated size of the buffer in bytes
  - *Length* number of bytes actually in use
- Use integer ranges [x,y] = { x, x+1, ..., y-1, y }
  - Size & 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
  len(s) $\leq$ alloc(s)

The Constraints

char s[n];	n $\subseteq$ alloc(s)
strcpy(dst,src)	len(src) $\subseteq$ len(dst)
p = strdup(s)	len(s) $\subseteq$ len(p) & alloc(s) $\subseteq$ alloc(p)
p[n] = '\0'	min(len(p),n+1)) $\subseteq$ len(p)

Constraint Solving

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

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
- Newer tools greatly improve on these results
  - E.g., METAL, Microsoft's SAL

Summary

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

The Last Slide

- Have a great New Year!