Language Security

Lecture 26

Lecture Outline

- Beyond compilers
  - Looking at other issues in programming language design and tools
- C
  - Arrays
  - Exploiting buffer overruns
- Java
  - Is type safety enough?

Platitudes

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

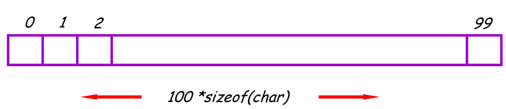
C Design Principles

- 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;
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

- The C code:

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

- The assembly code:

Regular C	C with bounds checks	
r1 = &buf1;	r1 = &buf1;	*Costly!*
r2 = load i;	r2 = load i;	
r3 = r2 * 4;	r3 = r2 * 4;	*Finding the array limits is non-trivial*
	if r3 < 0 then error;	
	r5 = load limit of buf1;	
	if r3 >= r5 then error;	
r4 = r1 + r3	r4 = r1 + r3	
store r4, 1	store r4, 1	

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)

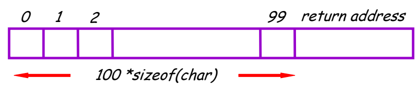
Buffer Overruns

- 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?
  - 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 */

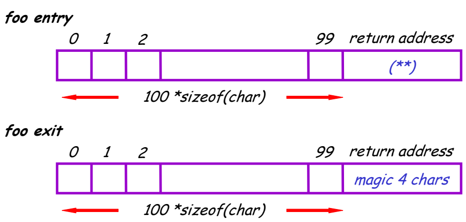


An Overrun Vulnerability

```
void foo(char in[]) {
  char buffer[100];
  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
    - 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

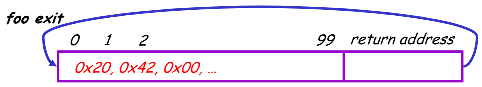
- We'll make the code jump to the following code:
- In C: exec("/bin/sh");
- In assembly (pretend):

```
    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

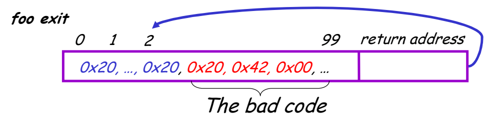
char in[104] = { 104 *magic chars* }
foo(in);



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

Guess the Location of the Injected Code

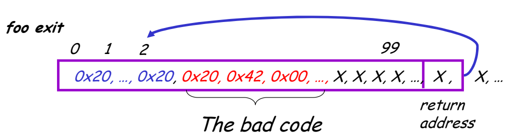
- Trial & error: gives you a ballpark
- Then pad the injected code with NOP
  - E.g. add $0, $1, 0x2020
    - stores result in $0 which is hardwired to 0 anyway
    - Encoded as 0x20202020



- 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 (Slammer)
- 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

Can Dataflow Analysis Help?

- Idea: for each variable used as an array index, calculate its possible range of values at each program point (eg. [0,99])
  - If we have array sizes, can check if bounds are respected
- Problem: infinite number of dataflow facts!
  - Analysis of loops probably won't terminate
- An efficient approximation gives too many false warnings
- Other compiler techniques more successful

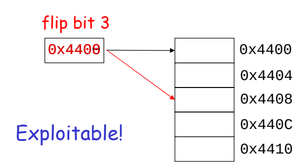
What about Java?

- Type safety prevents incorrectly-typed pointers
  - B b = new A() disallowed unless A extends B
- Array-bounds checks prevent buffer overflows
- Together, these checks prevent execution of arbitrary user code…

Unless the computer breaks!

Memory Errors

- A flip of some bit in memory
  - Can be caused by cosmic ray, or deliberately through radiation (heat)



Overview of Attack

- Step 1: use memory error to obtain two pointers p and q, such that p == q and p and q have incompatible, specially-designed static types
  - Normally prevented by Java type system
- Step 2: use p and q from Step 1 to write values into arbitrary memory addresses
  - Fill a block of memory with desired machine code
  - Overwrite dispatch table entry to point to block
  - Do the virtual call corresponding to modified entry

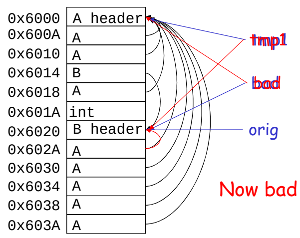
Special Classes For Attack

```
class A {
  A a1;
  A a2;
  B b; // for Step 1
  A a4;
  int i; // for address
         // in Step 2
}
```

```
class B {
  A a1;
  A a2;
  A a3;
  A a4;
  A a5;
}
```

Assume 3-word object header

Step 1 (Exploiting The Memory Error)



B orig;
A tmp1 = orig.a1;
B bad = tmp1.b;

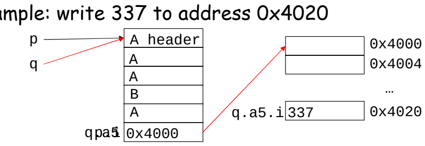
flip bit 5 in orig.a1

Now bad points to an A object!

Step 2 (Writing arbitrary memory)

```
A p; B q; // from Step 1, p == q; assume both point to an A
int offset = 8 * 4; // offset of i field in A
void write(int address, int value) {
  p.i = address – offset;
  q.a5.i = value; // q.a5 is an integer treated as a pointer
}
```

Example: write 337 to address 0x4020



Putting It All Together

```
A p; // pointer to single A object
while (true) {
  for (int i = 0; i < b_objs.length; i++) {
    B orig = b_objs[i];
    // Step 1, really check all fields
    A tmp1 = orig.a1;
    B q = tmp1.b;
    // See if we succeeded
    Object o1 = p; Object o2 = q;
    if (o1 == o2) {
      writeCode(p,q); // uses write from Step 2
    }
  }
}
```

- Heap has one A object, many B objects
- All fields of type A point to single object, to increase probability of success

Results (Govindavajhala and Appel)

- With software-injected memory errors, took over both IBM and Sun JVMs with 70% success rate
- Equally successful through heating DRAM with a lamp
- Defense: memory with error-correcting codes
  - ECC often not included to cut costs
- Most serious domain of attack is smart cards

Summary

- Programming language knowledge useful beyond compilers
  - Helps programmers understand the exact behavior of their code
  - Compiler techniques can help to address other problems like security (big research area)
- Safety and security are hard
  - Assumptions must be explicit