Announcements

• First project: Due: 25 Sept. 2006
  – [http://www.cis.upenn.edu/~cse331/project1.html](http://www.cis.upenn.edu/~cse331/project1.html)
  – Please e-mail savi@seas.upenn.edu with your project group by Weds. 13 Sept.
  – If you need a group, send e-mail

• Please put "cse331" in the subject of all course-related e-mail

• Prof. Zdancewic will be away Sept. 18 & 20.
  – Class will be taught by Peng Li
Plan for Today: Buffer Overflows

• Assigned Reading:
  Aleph One (1996)
  *Smashing the Stack for Fun and Profit*
  – This paper is essentially a tutorial for your project!

• Stack smashing is a particular (common) instance of a buffer overflow.
  – Easy to exploit in practice
Buffer Overrun in the News

• From Slashdot
  – “There is an unchecked buffer in Microsoft Data Access Components (MDAC) prior to version 2.7, the company said. MDAC is a "ubiquitous" technology used in Internet Explorer and the IIS web server. The buffer can be overrun with a malformed HTTP request, allowing arbitrary code to be executed on the target machine.”

  – http://www.theregister.co.uk/content/55/28215.html
The Consequences

• From Microsoft
  – “An attacker who successfully exploited it could gain complete control over an affected system, thereby gaining the ability to take any action that the legitimate user could take.”
Buffer Overflow Attacks

• > 50% of security incidents reported at CERT are related to buffer overflow attacks

• C and C++ programming languages don’t do array bounds checks.
  – Problem is *access control* but at a very fine level of granularity
3 parts of C memory model

- The code & data (or "text") segment
  - contains compiled code, constant strings, etc.
- The Heap
  - Stores dynamically allocated objects
  - Allocated via "malloc"
  - Deallocated via "free"
  - C runtime system
- The Stack
  - Stores local variables
  - Stores the return address of a function
C’s Control Stack

```c
f() {
    g(parameter);
}

g(char *args) {
    int x;
    // more local
    // variables
    ...
}
```

<table>
<thead>
<tr>
<th>ESP</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
</tr>
<tr>
<td>// local</td>
</tr>
<tr>
<td>// variables</td>
</tr>
</tbody>
</table>

Larger Addresses

- Base Pointer
- Return Addr.
- Input parameter
- f’s stack frame
C’s Control Stack

```c
f() {
    g(parameter);
}

g(char *args) {
    int x;
    // more local
    // variables
    ...
}
```
C’s Control Stack

```c
f() {
    g(parameter);
}

int x;
// local
// variables
// more local
// variables
...}
```

Larger Addresses

ESP

- Base Pointer
- Return Addr.
- Input parameter
- f’s stack frame
Buffer Overflow Example

```c
void g(char *text) {
    char buffer[128];
    strcpy(buffer, text);
}
```
Buffer Overflow Example

```c
char buffer[128];
strcpy(buffer, text);
```

Return Addr.

Attack code 132 bytes

Base Pointer

text

ADDR

ADDR: ?

f’s stack frame

Attack code 132 bytes

Return Addr.
Constructing a Payload

• Idea: Overwrite the return address on the stack
  – Value overwritten is an address of some code in the "payload"
  – The processor will jump to the instruction at that location
  – It may be hard to figure out precisely the location in memory

• You can increase the size of the "target" area by padding the code with no-op instructions

• You can increase the chance over overwriting the return address by putting many copies of the target address on the stack

[NOP]…[NOP]{attack code} {attack data}[ADDR]…[ADDR]
More About Payloads

• How do you construct the attack code to put in the payload?
  – You use a compiler!
  – Gcc + gdb + options to spit out assembly (hex encoded)

• What about the padding?
  – NOP on the x86 has the machine code 0x90

• How do you guess the ADDR to put in the payload?
  – Some guesswork here
  – Figure out where the first stack frame lives: OS & hardware platform dependent, but easy to figure out
  – Look at the program -- try to guess the stack depth at the point of the buffer overflow vulnerability.
  – Intel is little endian -- so if ADDR is: 0xbf9ae358 you actually need to put the following words in the payload: 0x58 0xe3 0x9a 0xbf
Finding Buffer Overflows

- The #1 source of exploitable vulnerabilities in software
- Caused because C and C++ are not safe languages
  - They use a “null” terminated string representation:
    ```
    "HELLO!\0"
    ```
  - Standard library routines assume that strings will have the null character at the end.
  - Bad defaults: the library routines don’t check inputs

- Easy to accidentally get wrong
- …even easier to maliciously attack
Buffer overflows in library code

• Basic problem is that the library routines look like this:

```c
void strcopy(char *dst, char *src) {
    int i = 0;
    while (src[i] != "\0") {
        dst[i] = src[i];
        i = i + 1;
    }
}
```

• If the memory allocated to `dst` is smaller than the memory needed to store the contents of `src`, a buffer overflow occurs.
Other common C/C++ errors

• Dereference NULL pointers.
  – Forgetting to check whether `malloc` succeeded

• Creating pointers to stack-allocated data structures
  – When stack frame is popped, the data structure is implicitly deallocated
  – When a new stack frame is pushed on, writing through the pointer can clobber its data

• Memory leaks
  – forgetting to free allocated memory

• Double free
  – Freeing the same data structure twice
If you must use C/C++

• Avoid the (long list of) broken library routines:
  – strcpy, strcat, sprintf, scanf, sscanf, gets, read, ...
• Use (but be careful with) the "safer" versions:
  – e.g. strncpy, snprintf, fgets, ...
• Always do bounds checks
  – One thing to look for when reviewing/auditing code
• Be careful to manage memory properly
  – Dangling pointers often crash program
  – Deallocate storage (otherwise program will have a memory leak)
• Be aware that doing all of this is difficult.
Tool support for C/C++

- Extensions to gcc that do array bounds checking
- Link against "safe" versions of libc (e.g. libsafe)
- Test programs with tools such as Purify, Splint, valgrind
- Compile programs using tools such as:
  - Stackguard and Pointguard (Cowan et al., immunix.org)

- Research compilers:
  - Ccured (Necula et al.)
  - Cyclone (Morrisett et al.)

- Binary rewriting techniques
  - Software fault isolation (Wahbe et al.)
Defeating Buffer Overflows

• Use a typesafe programming language
  – Java/C# are not vulnerable to these attacks
  – Garbage collection eliminates most memory management problems

• Some operating systems move the start of the stack on a per-process basis:
  – E.g. eniac-l
  – Doesn't provide complete protection (but it does make things harder)