• Exploiting software vulnerabilities is paramount to computer break-ins

• Common software vulnerabilities
  – input validation
  – buffer overflow
  – integer overflow
  – format string problems
  – interaction with environment variables
  – failure to handle errors
• **Buffer overflow** is a very common software vulnerability
  
  – the first major exploit was Morris Internet Worm in 1988 that exploited buffer overflow in fingerd
  
  – many others followed, such as Code Red worm in 2001
  
  – large percentage of all exploits in CERT vulnerability advisories describe buffer overflow or heap overflow problems
    
    • check US-CERT (United States Computer Emergency Readiness Team) alerts and ICS-CERT (Industrial Control Systems Cyber Emergency Response Team) advisories

• Buffer overflows often lead to total compromise of the host
A buffer overflow or buffer overrun is a condition under which more input can be placed in a buffer than the allocated capacity:

- the extra input for which there is no allocated memory overwrites other information
- the locations being overwritten could hold other variables, parameters, and control flow data such as return addresses

Developing buffer overflow attacks includes:

- locating buffer overflow within an application
- designing an exploit

Attacker needs to know which CPU and OS are running on the target machine.
• Simple example of buffer overflow in C

```c
void func(char *str1) {
    char str2[8];
    strcpy(str2, str1);
    printf("%s, %s\n", str1, str2);
}
```

– what happens if we call `func("abc")`?

– how about `func("reallylongstring")`?

• The problem occurs because `strcpy` doesn’t check the amount of data being copied
Not all languages are vulnerable

- some languages (Java, Python) provide strong type checking and have predefined operations on types
- they don’t permit storing more data than allocated space
- but safety comes at resource usage cost

Writing an exploit involves understanding process memory and stack layout

- in what direction the stack grows
- what data and in what order are placed on the stack
- how information is represented
• Generic process memory layout

- Stack
- Run time heap
- Global data
- Text (machine code)
Stack Buffer Overflow

- **Stack buffer overflow** or **stack smashing** occurs when the buffer is located on the stack.
- When a function is called, its data are placed on the stack:
  - this includes arguments, local variables, and return address.
- The **calling function** first:
  - pushes the parameters for the called function onto the stack:
    - normally in the reverse order
  - executes the call instruction which pushes the return address onto the stack.
Stack Buffer Overflow

- The called function
  - pushes the current frame pointer onto the stack
    - the frame pointer points to the calling routine’s stack
  - sets the frame pointer to be the current stack pointer value
  - allocates space for local variables
  - runs the called function
  - sets the stack pointer back to the value of the frame pointer
  - pops the old frame pointer
  - executes the return instruction which pops the return address off the stack giving control to the calling function

- The calling function pops parameters off the stack and continues
Suppose function \( f_1 \) calls \( f_2(\text{param1}, \text{param2}) \).
Let’s go back to our code example

```c
void func(char *str1) {
    char str2[8];
    strcpy(str2, str1);
    printf("%s, %s\n", str1, str2);
}
```

The stack will look like:

<table>
<thead>
<tr>
<th></th>
<th>str2</th>
<th>fptr</th>
<th>ret addr</th>
<th>str1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>←</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

top of stack

What if `str1` is 16 bytes long? After copying we get:

<table>
<thead>
<tr>
<th></th>
<th>str1</th>
<th>new addr</th>
<th>str1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>←</td>
<td>←</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

top of stack
• Now suppose that `str1` looks like this

```
<table>
<thead>
<tr>
<th>str1</th>
<th>ret addr</th>
<th>code for f</th>
</tr>
</thead>
</table>
```

– where program `f: exec("/bin/sh")`

• Now when function `func` exits, the user will be given a shell

• What happens
  – attack code runs on the stack
  – to determine the return address, attacker needs to guess position of the stack when `func` is called
The main problem with the above program was that there is no range checking in `strcpy()`

Some unsafe C library functions

- `strcpy(char *dest, char *src)`
- `strcat(char *dest, char *src)`
- `gets(char *str)`
- `scanf(const char *format, ...)`
- `printf(const char *format, ...)`
• As an example, consider attacking a web server
  – web server has a function with buffer overflow vulnerability which takes a URL
  – attacker can craft a long URL to obtain shell on web server

• Difficulties in constructing buffer overflow exploit
  – exploit code cannot contain the ‘\0’ character
  – overflow should not crash the program before the vulnerable function exits
Buffer Overflow

- How does one find buffer overflow vulnerabilities?
  - do it yourself
    - run the program (such as a web server) on your local machine
    - use long inputs with a specific pattern
    - if the program crashes, search core dump for the pattern to determine overflow location
  - use an existing automated tool
    - e.g., eEye Retina scanner, ISIC (IP Stack Integrity & Stability Checker)
- Once a vulnerability is found, use disassemblers and debuggers to construct an exploit
What can be done to defend against buffer overflow attacks?

Various mechanisms exist

- compile-time defenses
  - type safe language choice
  - static code analysis
  - safe libraries
  - stack protection

- run-time defenses
  - stack protection
  - address space randomization
• **Choice of programming language**
  
  – some languages (Java, ML) have a strong notion of variable type and define a set of permitted operations on them
  
  – they are not vulnerable to buffer overflow attacks
    
    • if they use external libraries written in an unsafe language, they can still be vulnerable
  
  – such languages are becoming increasingly popular
  
  – disadvantages
    
    • there is resource consumption cost at both compile time and run time
    
    • some functionality might be lost due to the distance from the architecture and machine language
Buffer Overflow Defenses

- **Static source code analysis**
  - statically check source code to detect buffer overflows
  - this allows to automate code review process
  - there are several consulting companies and several existing tools
    - Coverity, Microsoft PREfix and PREfast, etc.
  - find many bugs, but not all

- **Safe coding practices**
  - buffer overflows can be prevented by handling errors gracefully
  - first check buffer size to ensure that sufficient space has been allocated
  - more complex data structure require additional care
Example problems that can be detected through static source code analysis:

- null pointer dereference
- use after freeing
- double freeing
- array indexing errors
- mismatched array on create/delete
- potential stack overrun
- potential heap overrun
- returning pointer to local variables
- logically inconsistent code
- uninitialized variables
- invalid use of negative values
- underallocations of dynamic data
- memory leaks
- file handle leaks
- network resource leaks
- unused values
- unhandled return values
- use of invalid iterators
Buffer Overflow Defenses

- **Use of safe libraries**
  - replace standard unsafe library routines with a safer version
  - **Libsafe** (by Avaya Labs) is a well-known example
    - dynamically loaded library
    - ensures that copy operations don’t extend beyond the current stack frame
    - e.g., in `strcpy(dest, src)`, if \( \text{loc}(fp) - \text{loc}(dest) > \\text{strlen}(src) \), permit copy; otherwise, terminate the application

- **Language extensions** for adding range checking also exist
Buffer Overflow Defenses

• Stack protection mechanisms
  – mark stack segment as non-executable
    • this will prevent several types of buffer overflow attacks
    • this is supported in SP2, code patches exist for Linux, Solaris
  • there are disadvantages
    – it does not prevent all types of overflow exploits
    – some applications need executable stack (e.g., LISP interpreters)
  – disallow changes to the stack frame during function execution
    • instruct the function entry and exist code to check the stack for corruption
    • if any modification is found, abort the program
• **StackGuard** is one of the best known stack protection mechanisms
  
  – insert a canary value between the old frame pointer address and local variables
  
  – the entry code places a canary, and the exit code checks it for corruption

<table>
<thead>
<tr>
<th>←</th>
<th>local vars</th>
<th>canary</th>
<th>fpotr</th>
<th>ret addr</th>
<th>params</th>
<th>→</th>
<th>top of stack</th>
</tr>
</thead>
</table>

  
  – random canary
  
  • the canary value is chosen at random during program startup
  
  • it is known to all functions, but unpredictable to attacker
• **StackGuard** (cont.)
  
  – terminator canary
    
    • set canary value to ‘\0’ (or an equivalent terminator value)
    
    • string functions will not copy beyond the terminator
    
    • attacker cannot use string functions to corrupt stack
  
  – StackGuard can be used as a GCC extension
    
    • performance overhead is minimal: 8% for Apache web server
    
    • similar functionality exists for Windows
    
    • newer version **PointGuard** offers protection against more types of overflow exploits
• **StackGuard** (cont.)
  – disadvantages
    • all programs that require protection need to be recompiled
    • can cause problems with other programs such as debuggers
    • some stack smashing attacks can leave canaries untouched

• **StackShield**
  – stack frame is protected without altering the stack with canaries
  – on function entry, added code writes a copy of the return address to a safe memory region
  – the exit code compares the return address with the stored value and aborts the program in case of corruption
  – StackShield is also available as a GCC extension
**Randomization**

- buffer overflow exploits need to know (virtual) address to which to pass control
  - address of attack code in the buffer
  - address of library routines for return-to-libc attack
- the same address is used on many machines
  - Slammer worm infected 75,000 MS SQL servers using the same code on every machine
- the idea is to introduce artificial diversity
  - make stack and other addresses unpredictable and different from one machine to another
Buffer Overflow Defenses

• **Address space randomization**
  - arrange key data areas randomly in address space of a process
    - e.g., positions of heap, stack, libraries
  - correct address guessing is significantly more difficult
  - support for this defense exists in many operating systems

• **Instruction set randomization**
  - each program has a different and secret instruction set
  - uses translator to randomize instructions at load time
  - attacker no longer can execute her own code
  - what constitutes the instruction set depends on the environment
Other Types of Overflow Attacks

- **Frame pointer replacement**
  - the attack overwrites the buffer and stored frame pointer
  
  
  ┌─┬─┬─┬─┐
  │ str1 │ new fpt │ ret addr │ str1 │
  └─┬─┬─┬─┘
       top of stack

  - the buffer contains a dummy stack frame with a return address pointing to the shellcode in the same buffer
  - this can be used when only a limited buffer overflow is possible

- **Off-by-one attack**
  - as a variant of the above attack, a programming error might permit copying just one byte more than the available space
  - this happens when the size of the buffer is checked incorrectly
Other Types of Overflow Attacks

• **Off-by-one attack** (cont.)
  
  – example code:

  ```c
  void f(char *str) {
      char buf[16];
      if (strlen(str) <= sizeof(buf)) {
          strcpy(buf, str);
      }
  }
  
  – only one byte of the frame pointer can be overwritten

  ![](image.png)

  • it corresponds to the least significant byte on x86 architectures
  
  • a one byte change might be enough!
Other Types of Overflow Attacks

- **Return to system call attack**
  - also known as return-to-libc attack
  - assume that the stack is made non-executable as a protection mechanism
  - a new type of attack overwrites the return address to jump to existing code
    - `system()` call in **libc** is most commonly used
  - what needs to be done
    - overwrite frame pointer with a suitable value
    - replace return address with address of the library function
    - write a value that the library function will believe is return address
Other Types of Overflow Attacks

• Return to system call attack (cont.)
  – finally, specify parameters to the library function
    • i.e., call the shell

    ![Diagram showing stack layout](image)

    system() in libc

    str1  |  ret addr  |  fake ret addr  |  top of stack

    "/bin/sh"

  – what happens?
    • when the attacked function returns, it transfers control to the return address, which calls the library function
    • the library function treats the value on top of the stack as a return address and the values before as its parameters
Other Types of Overflow Attacks

• Heap overflow
  – with stack protection techniques, attackers started exploiting overflows in non-stack buffers
  – the heap that stores dynamically allocated data structures is such a target
  – if heap contains a buffer vulnerable to overflow, other data on the heap may be overwritten
  – heap doesn’t contain return addresses to transfer controls, but can contain pointers to functions
    • attacker can overwrite the pointer to reference code in the same buffer that calls shellcode
    • if the function is called, the attack can succeed
Other Types of Overflow Attacks

- **Heap overflow** (cont.)
  - defenses include making heap non-executable and heap address randomization

- **Other types of overflow**
  - similar vulnerabilities exist for static (global) data
  - such data is allocated in different space, but the treats remain similar to heap overflow
  - finally, there are also integer overflow and format string overflow attacks
• Additional articles
  – “Smashing the stack for fun and profit” by Aleph One, 1996
  – “Buffer overflows: attacks and defenses for the vulnerability of the
decade” by Cowan et al., 2000
  – “Bypassing non-executable-stack during exploitation using
return-to-libc” by c0ntex, 2005

• What is next
  – other types of software vulnerabilities
  – input validation, environment variables, race conditions, etc.