Buffer Overflows
Objectives

- Describe buffer overflow
- List types of buffer overflows
- Identify techniques used to cause a buffer overflow
- Comprehend techniques used to detect buffer overflow conditions
- Understand methods used in preventing buffer overflows
Buffer Overflows

• Buffer overflow
  – Condition common to structured programming languages such as the “C” language
  – Happens when input applied to a variable is larger than the memory allotted to that variable

• When an attacker sends input in excess of the expected range of the value
  – The target system will either crash or execute the malicious code sent by the attacker
Standard Execution of a C program

- **main function**
  - Entry point to the detailed code in the application
  - Responsible for calling other functions
    - Each of which executes a particular task, and may call other functions in turn
- Functions use variables to store values that may be stored temporarily or permanently
- Once a function has completed, the program control returns to the calling or invoking function
Standard Execution of a C program (continued)

- Buffer overflow bug targets the variables that are used by functions to store values
- Variables are assigned a fixed memory space to store the data
- Buffer overflow has a goal of overloading the memory space provided to the variable
  - Extra characters are stored in a memory space that is not assigned to the variable
Figure 12-1  Simplified buffer overflow
Standard Execution of a C program (continued)

• Function with an overloaded variable is not able to determine the function that called it
  – Could result in the crashing of the program
• Hackers are able to manipulate the value provided to the variable
  – So it is stored in a specific memory space to execute some predetermined malicious code
• Buffer overflows are not always intentional attacks
Standard Execution of a C program (continued)

• Avoiding buffer overflows
  – Check to see that no value greater than the memory assigned to the variable is specified for it
  – Define the sequence of steps that the program has to follow in case of a buffer overflow

• Executable space protection
  – On some specific operating systems, the kernel can be patched in such a way that running processes are not affected by buffer overflow conditions
Types of Buffer Overflows

- Buffer overflows can be divided into two categories: stack overflow and heap overflow
Stack Overflow

- Programs use a memory stack area to store values for variables
- Stack is intended to ensure that there is sufficient memory space for all functions to operate
- Occasionally the stack is insufficient to complete the functions and an error is generated
- Stack stores details regarding the function that called the currently executing function
  - Information can be lost after the stack is corrupted
Stack Overflow (continued)

• Hackers write the code for a buffer overflow in such a manner that
  – The code to which a function’s pointers are indicating is code of the hacker’s choosing

• Process of an exploit
  – Hacker searches for a chance to overflow the buffer
  – Hacker determines memory assigned to the variable
  – Hacker specifies a value greater than the maximum capacity of the variable
  – Variable takes the value
Stack Overflow (continued)

• Hacker checks for some specific functions to ascertain the possibility of a buffer overflow
  - strcpy
  - scanf
  - fgets
  - wstrcpy
  - wstrncat
  - sprintf
  - gets
  - strcat
Heap Overflows

• A heap is similar to a stack
  – Provides memory to the application various functions
  – Provides a permanent memory space
• Data stored in a heap can be used throughout various functions and commands
• A heap is randomly accessed because it stores values statically
• Size of a heap usually grows as new variables’ values are introduced
Heap Overflows (continued)

<table>
<thead>
<tr>
<th>Differentiation Points</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Use</td>
<td>Calling Functions</td>
<td>Long-term storage for data</td>
</tr>
<tr>
<td></td>
<td>Short-term storage of variables</td>
<td></td>
</tr>
<tr>
<td>Access</td>
<td>Frequently accessed</td>
<td>Randomly accessed</td>
</tr>
<tr>
<td>Expansion</td>
<td>Automatically</td>
<td>Automatically and using <code>malloc()</code> and <code>brk()</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence of value storage</td>
<td>LIFO</td>
<td>--</td>
</tr>
<tr>
<td>Grows</td>
<td>Grows from higher to lower addresses</td>
<td>Grows from lower to higher addresses</td>
</tr>
</tbody>
</table>
Figure 12-2  Structure of stacks and heaps
Heap Overflows (continued)

- Heap overflow is known as the corruption of the instruction pointer
  - Instruction pointer points to the memory area where the function to be executed is stored
More Methods for Causing a Buffer Overflow

• Traditional methods include
  – Providing input values that are greater than the memory allocated for a variable
• This section details two other methods:
  – Character-set decoding
  – Nybble-to-byte compression
Character-Set Decoding

- Uses the characters that are read differently by the computer and acquire larger space
- Additional bytes of data may cause the input value to exceed the memory limitation of the variable
- Applicable to situations in which the user specifies a value from an HTML page
- Becomes a weakness whenever a back-end script reads the code
  - And, after expanding the value, results in a buffer overflow
### Table 12-2  Character conversions

<table>
<thead>
<tr>
<th>Character Sign</th>
<th>Read As</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverted exclamation sign</td>
<td>&amp;#161</td>
</tr>
<tr>
<td>Cent sign</td>
<td>&amp;#162</td>
</tr>
<tr>
<td>Pound sign</td>
<td>&amp;#163</td>
</tr>
<tr>
<td>Currency sign</td>
<td>&amp;#164</td>
</tr>
<tr>
<td>Section sign</td>
<td>&amp;#167</td>
</tr>
<tr>
<td>Copyright sign</td>
<td>&amp;#169</td>
</tr>
<tr>
<td>Degree sign</td>
<td>&amp;#176</td>
</tr>
<tr>
<td>Right-pointing double angle quotation marks</td>
<td>&amp;#187</td>
</tr>
<tr>
<td>Latin capital letter A with grave</td>
<td>&amp;#192</td>
</tr>
</tbody>
</table>
Figure 12-3   Output of HTML form code
Character-Set Decoding (continued)

• Size of the stack is computer-dependent
  – Buffer overflow may not happen until a value that exceeds the stack size is specified
• Buffer overflow may occur when verification for the length of the input value is made at the client-side
  – And the back-end script accepts it without performing checks
• Double validation
  – Indicates to potential hackers that buffer overflow exploits are not possible on your Web site
Character-Set Decoding (continued)

Figure 12-4  Input as read by back-end script
Nybble-to-Byte Compression

• Compression of data that is passed as input value to variables of the function that might be overflowed
• Method uses the buffer in a more efficient manner with a higher amount of data
• Focus is to minimize code size
  – So hackers can double the amount of code in buffer
Buffer Overflows: Detection and Prevention

• Identify programming practices and functions that are potentially vulnerable to buffer overflow
Detecting Buffer Overflow

• Identify the functions and variables that can lead to buffer overflows
  – Check the reaction of the application whenever a large set of character data is supplied to a variable
• Function may include length verification
  – And thus return an error message if the data exceeds the expected size
• Can be a painstaking, tedious process
  – Because all variables that accept values must be checked
Detecting Buffer Overflow (continued)

• Precaution should be taken to ensure that the input data is provided in the correct format

• For interactive Web pages
  – Consider that the hidden data is also a part of the input that is given to the string

• When specifying the input data
  – It is important to check that no NULL characters (empty fields) are being passed
Preventing Buffer Overflow

• After a buffer overflow exploit has been detected
  – The probability of its existence in other applications by the same vendor is higher
• Bug is typically fixed by programming the functions to perform an input validity check
• Providing a null terminator will prevent the buffer overflow even
  – If additional values have been specified
Preventing Buffer Overflow (continued)

• Consider developing specific programming guideline policies for your organization
  – Having, understanding, and applying secure-coding best practices may not be entirely foolproof
• Options are available to avoid the use of function calls that are vulnerable to buffer overflows
Preventing Buffer Overflow (continued)

<table>
<thead>
<tr>
<th>Unsafe Function Call</th>
<th>Safe Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>gets()</td>
<td>fgets()</td>
</tr>
<tr>
<td>strcpy()</td>
<td>strncpy()</td>
</tr>
<tr>
<td>strcat()</td>
<td>strncat()</td>
</tr>
<tr>
<td>sprintf()</td>
<td>snprintf()</td>
</tr>
</tbody>
</table>
Preventing Buffer Overflow (continued)

• Checks must be made to validate the input values in both the new and old applications
• Software can be installed to keep a continuous check on a buffer overflow condition
  – Software must be updated with all available security patches