Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns

Jonathan Pincus (MSR) and Brandon Baker (MS)
Buffer Overflows and How they Occur

• Buffer is a contiguous segment of memory of a fixed size

• Program tries to write beyond the boundary of the buffer

• C/C++ are not type safe, so any memory location can be interpreted as any type of data

• No boundary checking of arrays when data is written

• Data is written into adjacent memory locations beyond the boundary of the array and is the payload

• Payload can contain opcodes which are interpreted by the program as code and the program’s flow of control is changed (Or the program will most likely just crash)
A Simple Buffer Overflow

- `char buffer[5];
  strcpy(buffer, "ABCDEF");`

- **View of memory after allocation**

  |-----------|-----------|-----------|-----------|-----------|

- **After copying the string**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>\0</th>
</tr>
</thead>
</table>

- The F and \0 are in adjacent memory locations
Convention I’m Going to Use

• Virtual address space is 4 GB 0x00000000 to 0xFFFFFFFF

• Stack starts at the bottom of the page and grows towards lower addresses
Function Calls – The Stack

• The parameters to the function are pushed onto the stack

• The current point of execution, i.e. – the instruction pointer (EIP) is pushed

• Points to the next instruction, i.e. – the instruction to be performed after the function returns

• EBP is pushed onto the stack

• Local variables of the functions
Function Calls – The Stack

• Consider the function –

```c
void function(int A, int B) {
    int C;
}
```

• The stack will look like

```
  SP
  B
  A
  EIP
  EBP
  C
```

Stack Smashing

• The local function is above the EBP and EIP on the stack

• Buffer overflows over the EBP and the EIP

• Saved EIP is overwritten with an address of the attacker’s choice

• Execution path returns to the value pointed by EIP, which could be some custom code
Before and After an Attack

Local Variables

0x00000000

0xB49E7884

Arguments

0xFFFFFFFF

buff[0] : 0x90
......
buff[n] : exit()
D : 0x90909090
C : 0x90909090
EBP : 0x90909090
EIP : 0xB49E7884
A
B

buff[0] : 0x90
......
buff[n] : exit()
D : 0x90909090
C : 0x90909090
EBP : 0x90909090
EIP : 0xB49E7884
A
B
Buffer Overflow Protection Mechanisms

• Number of mechanisms have come to protect against buffer overflows

• Try and detect the buffer overflows or prevent the execution of the shellcode after the overflow

• Most common amongst these are
  • Non executable stacks
  • Canary Values (Stackguard)
  • Address Space Layout Randomization (ASLR)
Non-Executable Stacks

• NX stack

• Marks certain areas of memory as non-executable

• Processor will refuse to execute instructions if EIP points into these locations

• Hardware or software implemented

• In Windows, the feature is called Data Execution Prevention or DEP
Canary Values

• Value is stored and written in between the buffer and control variables such as EBP and EIP on the stack

• Canary can be a random value stored in an arbitrary location in an unmapped page

• During a buffer overflow the canary value is overwritten before the EIP

• While returning from the function, the canary is checked

• If it has been overwritten, the program faults
Arc Injection Attack

• EIP is overrun to point to a location already in the program’s address space

• system() takes an argument R and makes a system call to create the process

• R can be made to point to an attacker pointed string

• Since system is in libc which is automatically linked into all applications, also called return-to-libc attacks

• As code is on an executable page, NX is bypassed
Chaining Arc Injection Attacks

- Shell can be obtained by running the system() command with ‘bin/sh’ as an argument.

- When the called function has finished executing, it will look for a return address on the stack.

- By providing the location of another function, we can get it to jump there and chain a number of function calls together.

- The address of exit() is passed as the return for system() so that the program exits after executing and giving us a shell.
Arc Injection Attack

• The buffer is filled with garbage

• EIP is overwritten with the address of system()

• Following this is the address of exit(), which acts as the EIP after the system() call returns

• Following this is the address of ‘bin/sh’
Function Pointer Clobbering

• Modify a function pointer to point to attacker supplied code

• When the pointer is used, it calls the modified code as opposed to the original code

```c
void f2a(void *arg, size_t len)
{
    char buf[100];
    void (*f)() = ....;
    memcpy(buf, arg, len);
    f();
}
```

• Can bypass both NX and Canary values protection
Data Pointer Modification

• Occurs when there is a pointer that is used for assignment operations after the buffer overflow

• By writing a location into the pointer, one can write to any arbitrary memory location

• Can be used to modify an external function pointer, or also to overwrite the global canary value

• When the function is called, or the function is returning and comparing canary values, it uses the attacker modified value

```c
long val, *ptr;
extern void (*f)();
memcpy(buff, arg, len);
*ptr = val;
```
Exception Handler Hijacking

• Windows has an exception handling mechanism called Structured Exception Handlers (SEH)

• Program has a registered list of exceptions that are stored in a linked list

• When an exception occurs, one or more of these is invoked by calling the function pointer present on the list

• Exception handler of the entire Thread Environment Block can be replaced

• Exception Handlers contain two elements
  • Handler – code to handle the exception
  • NextSEH – pointer to the next exception handler
SEH Overflow

• The handler is filled up with the address of POP, POP, RET instructions to trampoline to a known location

• NextSEH is overflowed to contain a 6 byte jump to jump over the opcode and the pointer to POP, POP, RET

• Shellcode starts immediately after the SEH handler
VPTR Smashing

- C++ compilers implement a virtual function table (VTBL) with each class, which contains an array of function pointers called virtual pointers (VPTR)

- VPTR smashing replaces a VPTR in the VTBL with a user specified value

- This function is now called whenever that virtual function is invoked on the object

- Can happen when the object is allocated either on the stack or on the heap
Heap Smashing

• Heap smashing exploits the implementation of the dynamic memory allocation routines

• Some allocators keep headers in allocated and freed blocks which are chained together in doubly linked lists

• These are updated during malloc and free operations

• A buffer overflow can corrupt the pointers in the header of an adjacent block, which can lead to arbitrary code execution when the free is called on the adjacent block

• More difficult than stack smashing attacks, but have been commoditized recently
Conclusions

• The volume of existing low level code in C/C++ means that buffer overflows are likely to remain a major source of vulnerabilities.

• Newer techniques give lie to the assumption that buffer overflows necessarily inject shellcode or that they have to modify the saved return address.

• New language functionality like virtual functions result in new avenues for buffer overflow attacks.

• “Arms-race” between attackers and defenders is likely to continue.
Questions

Language Support

• Can we have languages that take care of vulnerabilities and prevent buffer overflows? Or is it impossible?

• How do type-safe languages prevent buffer overflows?

• Will changing the environment of languages (like using compiler extensions and safe versions of unsafe functions) prevent buffer overflows?

• Early attempts to move OSes from assembly to C were met with skepticism. Should we make a shift to a higher level language for operating systems where the exploits in the paper are not possible?
Questions

Prevention

• Wouldn’t carefully written and audited code prevent it?

• Does sandboxing the entire system like VM’s work? Can we prove that there will be no attack past the protection boundary?

• Arc injection seems particularly scary. How do we prevent it?

• Can loading all function pointers and return addresses at compile time and marking them read only help?
Questions

Miscellaneous

• Can these types of attacks affect the kernel?

• Is there any way around this arms race that they mention in the conclusion?