Riding the Overflow – Then and Now

Miroslav Štampar
(mstampar@zsis.hr)
I USED TO BE GOOD AT BUFFER OVERFLOWS

UNTIL THEY STARTED TO PROTECT
Buffer overflow

- (a.k.a.) Buffer overrun
- An anomaly where a program, while writing data to the buffer, overruns its boundary, thus overwriting adjacent memory
- Commonly associated with programming languages C and C++ (no bounds checking)
- Stack-based (e.g. statically allocated built-in array at compile time) – overwriting stack elements
- Heap-based (e.g. dynamically allocated malloc() array at run time) – overwriting heap internal structures (e.g. linked list pointers)
Stack-based overflow
Heap-based overflow
Vulnerable code (stack-based)

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    char buff[100];
    if (argc >= 2) {
        strcpy(buff, argv[1]);
    }
    return 0;
}
```
Vulnerable code (heap-based)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *p, *q;
    p = malloc(1024);
    q = malloc(1024);
    if (argc >= 2)
    {
        strcpy(p, argv[1]);
    }
    free(q);
    free(p);
    return 0;
}
```
History

- 1961 - Burroughs 5000 (executable space protection)
- 1972 - Computer Security Technology Planning Study (buffer overflow as an idea)
- 1988 - Morris worm (earliest exploitation – \texttt{gets() in fingerd})
- 1995 - Buffer overflow rediscovered (Bugtraq)
- 1996 - “Smashing the Stack for Fun and Profit” (Aleph One)
- 1997 - “Return-into-lib(c) exploits” (Solar Designer)
- 2000 - The Linux PaX project
- 2001 - Code Red (IIS 5.0); Heap spraying (MS01-033 – Index Server ISAPI Extension)
- 2003 - SQL Slammer (MsSQL 2000); Microsoft VS 2003 flag /GS
- 2004 - NX on Linux (kernel 2.6.8); DEP on Windows (XP SP2); Egg hunting (skape)
- 2005 - ASLR on Linux (kernel 2.6.12); GCC flag \texttt{-fstack-protector}
- 2007 - ASLR on Windows (Vista); ROP (Sebastian Krahmer)
**DEP/NX**

- Data Execution Prevention/No eXecute
- (a.k.a.) Non-executable stack, Execute Disable, Exec Shield (Linux), W^X (FreeBSD)
- Set of hardware and software technologies that perform additional checks on memory
- Provides protection for all memory pages that are not specifically marked as executable
- Processor must support hardware-enforced mechanism (NX/EVP/XD)
- Executables and libraries have to be specifically linked (problems with older software)
ASLR

- Address Space Layout Randomization
- Introduces the randomness into the address space of process
- Positions of key data areas are randomly scattered (i.e. dynamic/shared libraries, heap and stack)
- Its strength is based upon the low chance of an attacker guessing the locations of randomly placed areas
- Executables and dynamic/shared libraries have to be specifically linked (problems with older software)
Stack canaries

- (a.k.a.) Stack cookies, Stack-Smashing Protector (SSP)
- Named for analogy to a canary in a coal mine
- Implemented by the compiler
- Placing a small (e.g. random) integer value to stack just before the return pointer
- In order to overwrite the return pointer (and thus take control of the process) the canary value would also be overwritten
- This value is checked to make sure it has not changed before a routine uses the return pointer from the stack
ASCII armor

- Generally maps important library addresses (e.g. libc) to a memory range containing a NULL byte (e.g. `0x00****** - 0x0100******`)
- Makes it hard to construct address or pass arguments by exploiting string functions (e.g. `strcpy()`)  
- Not effective when NULL byte is not an issue
- Easily bypassable by using PLT (Procedure Language Table) entries in case of position independent binary
SEH

- Structured Exception Handler
- Implemented by the compiler
- Pointer to the exception handler is added to the stack in the form of the “Exception Registration Record” (SEH) and “Next Exception Registration Record” (nSEH)
- If the buffer is overflown and (junk) data is written to the SEH (located eight bytes after ESP), invalid handler is called due to the inherently raised exception (i.e. STATUS_ACCESS_VIOLATION), thus preventing us from successful execution of our payload
SEH (chain)

```
N H → app!_except_handler4
N H → k32!_except_handler4
N H → ntdll!FinalExceptionHandler
```
SEHOP

- Structured Exception Handler Overwrite Protection
- Blocks exploits that use (highly popular) SEH overwrite method
- Enabled by default on Windows Server 2008, disabled on Windows Vista SP1 and Windows 7
- Symbolic exception registration record appended to the end of exception handler list
- Integrity of exception handler chain is broken if symbolic record can't be reached and/or if it's found to be invalid
SafeSEH

- Safe Structured Exception Handling
- (a.k.a.) Software-enforced DEP
- All exception handlers' entry points collected to a designated read-only table collected at the compilation time
- Safe Exception Handler Table
- Attempt to execute any unregistered exception handler will result in the immediate program termination
Safe functions

- Well-written functions that automatically perform buffer management (including bounds checking), reducing the occurrence and impact of buffer overflows
- Usually by introducing explicit parameter size

<table>
<thead>
<tr>
<th>Unsafe</th>
<th>Safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>sprintf</td>
<td>snprintf, asprintf</td>
</tr>
<tr>
<td>strncat</td>
<td>strlcat</td>
</tr>
<tr>
<td>gets</td>
<td>fgets</td>
</tr>
<tr>
<td>strcat</td>
<td>strlcat</td>
</tr>
<tr>
<td>vsprintf</td>
<td>vsprintf, vasprintf</td>
</tr>
<tr>
<td>strcpy</td>
<td>strlcpy</td>
</tr>
<tr>
<td>strncpy</td>
<td>strlcpy</td>
</tr>
</tbody>
</table>
**NOP sled**

- (a.k.a.) NOP slide, NOP ramp
- Oldest and most widely known method for stack buffer overflow exploitation
- Large sequence of NOP (no-operation) instructions meant to "slide" the CPU's execution flow
- Used when jump location has to be given (payload), while it's impossible to be exactly predicted

\[ |\text{buffer}| = |\text{NOP sled}| + |\text{payload}| + |\text{guessed address from inside NOP sled (EIP)}| \]
ret2libc

- (a.k.a.) ret2system, arc injection
- Overwriting the return address with location of a function that is already loaded in the binary or via shared library
- Also, providing required arguments through stack overwrite
- Shared library libc is always linked to executables on UNIX style systems and provides useful calls (e.g. system())

|buffer| = |junk| + |address of function system() (EIP)| + |address of function exit()| + |address of string “/bin/sh”|
**ret2reg**

- Return-to-register (e.g. ESP, EAX, etc.)
- (a.k.a.) Trampolining
- Also, variants like ret2pop, ret2ret, etc.
- We overwrite the EIP with the address of an existing instruction that would jump to the location of a register
- Preferred choice is the register pointing to the location inside our buffer (usually ESP)
- Much more reliable method than NOP sled

\[ \text{buffer} = \text{junk} + \text{address of JMP ESP or CALL ESP instruction (EIP)} + \text{compensating NOPs} + \text{payload} \]
**Egg hunting**

- Used in reduced buffer space situations
- Allows usage of a small payload ("egg hunter") to find the actual (bigger) payload
- The final payload must be somewhere in memory (stack, heap or secondary buffer)
- Final payload must be prepended with the unique marking string (2x4 bytes) called "egg"
- Egg hunter types: SEH, IsBadReadPtr, NtDisplayString, NtAccessCheckAndAuditAlarm

|buffer| = |junk| + |egg hunter| + |address of JMP ESP or CALL ESP instruction (EIP)| + |JMP to egg hunter| + |junk| + |egg| + |egg| + |payload|
Egg hunter (NtDisplayString)

```assembly
loop_inc_page:
or    dx, 0x0fff       // Add PAGE_SIZE-1 to edx
loop_inc_one:
ic    edx              // Increment our pointer by one
loop_check:
push   edx              // Save edx
push   0x43             // Push NtDisplayString
pop    eax              // Pop into eax
int    0x2e             // Perform the syscall
cmp    al, 0x05         // Did we get 0xc0000005 (ACCESS_VIOLATION) ?
pop    edx              // Restore edx
loop_check_8_valid:     // Yes, invalid ptr, go to the next page
je     loop_inc_page    // Yes, invalid ptr, go to the next page
is_egg:
mov    eax, 0x50905090  // Throw our egg in eax
mov    edi, edx         // Set edi to the pointer we validated
scasd                              // Compare the dword in edi to eax
jnz    loop_inc_one     // No match? Increment the pointer by one
scasd                              // Compare the dword in edi to eax again (which is now
edx + 4)
jnz    loop_inc        // No match? Increment the pointer by one
matched:
jmp    edi              // Found the egg. Jump 8 bytes past it into our code.
```
SEH bypass

■ SEH is highly flawed against buffer overflows
■ Overwrite (last in chain) SEH with address of "POP; POP; RET" sequence of instructions and nSEH with explicit relative "JMP" to payload
■ Deliberate exception has to be caused (inherently by sending malformed buffer)
■ “POP; POP; RET” passes the execution flow to the nSEH's JMP, which afterwards jumps to the payload at the end of the buffer

\[ |\text{buffer}| = |\text{junk}| + |\text{JMP to payload (nSEH)}| + |\text{address of “POP; POP; RET” sequence of instructions (SEH)}| + |\text{compensating NOPs}| + |\text{payload}| \]
ROP

- Return-Oriented Programming
- Attacker executes carefully chosen machine instruction sequences called “gadgets”
- Each gadget ends with an instruction RET (e.g. “INC EAX; RET”)
- ROP “chain” consists of gadget memory locations
- Provides a fully functional language that can be used to perform any operation desired (usually to disable DEP)

|buffer| = |junk| + |ROP chain to disable DEP| + |compensating NOPs| + |payload|
# ROP (disable DEP)

<table>
<thead>
<tr>
<th>API / OS</th>
<th>XP SP2</th>
<th>XP SP3</th>
<th>Vista SP0</th>
<th>Vista SP1</th>
<th>Windows 7</th>
<th>Windows 2003 SP1</th>
<th>Windows 2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>VirtualAlloc</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>HeapCreate</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SetProcessDEPPolicy</td>
<td>no (1)</td>
<td>yes</td>
<td>no (1)</td>
<td>yes</td>
<td>no (2)</td>
<td>no (1)</td>
<td>yes</td>
</tr>
<tr>
<td>NtSetInformationProcess</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no (2)</td>
<td>no (2)</td>
<td>yes</td>
<td>no (2)</td>
</tr>
<tr>
<td>VirtualProtect</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>WriteProcessMemory</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

(1) = doesn't exist  
(2) = will fail because of default DEP Policy settings

Taken from: https://www.corelan.be
Heap spray

- Top payload delivery method used in browser exploits (and recent high profile attacks)
- Takes advantage of the fact that the heap management is deterministic
- Attacker needs to be able to deliver the payload in the right location in memory before triggering the bug that leads to EIP control
- A good heap spray (if done right) will end up allocating a chunk of memory at a predictable location, after a certain amount of allocations
- At the end (predictable) heap address needs to be put into EIP
Heap spray (memory)
Demo time
Questions?