Protection Against Exploitation of Stack And Heap Overflows

Yinrong Huang

Exurity Inc., Canada

We welcome you to utilize the concept presented in this paper. We also sincerely appreciate your generous financial contribution to our research.

For updated info, please check
http://members.rogers.com/exurity/

Protection Against Exploitation of Stack And Heap Overflows

Yinrong Huang
Exurity Inc., Canada

1 Introduction

Reports of buffer overflow vulnerability pop up almost everyday on SecurityFocus' buqtraq mailing lists (1). Vendors tried, sometimes desperately, to patch up their software holes before their software vulnerabilities were maliciously exploited.

My previous article describes a mechanism to protect against the malicious exploitation, like SQL Slammer worm, of overflowed stack by inserting a breakpoint opcode 0CCh onto the stack before RET (2). This article addresses how to protect against off-by-one exploitation of overflowed stack, how to repair the heap allocation and free scheme to catch the exploitation, potentially malicious, of the overflowed heap (3), how to protect against overflowed structured exception handling (SEH) frame like Code Red (5), and how to refuse RET to defend against brute force exploitation shown in WebDAV exploitation (6).

2 Protection Against Off-by-one Exploitation

Upon entering a called C function on x86-based computers, execution of code usually begins with prolog code such as:

```
Push EBP
Mov  EBP, ESP
```

Before leaving the called C function, execution of code usually ends with epilogue code such as (or opcode LEAVE):

```
Mov  ESP, EBP
Pop  EBP
RET
```

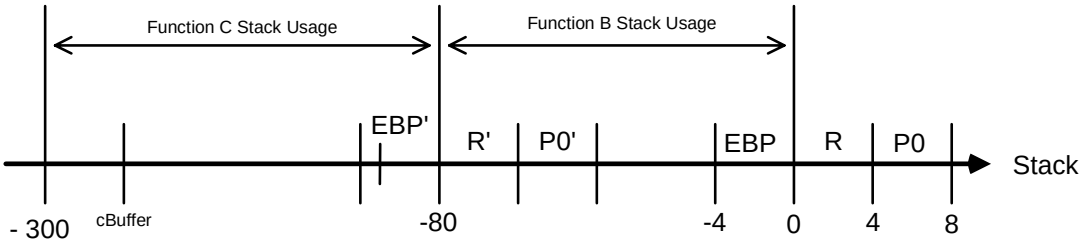

Figure 1 Normal Stack Usage For Two Function Calls with "mov esp, ebp; pop ebp" epilogue scheme Notice EBP' = - 4

Figure 1 illustrates the normal stack usage for a serial C code below. Without deliberate overflowing of the cBuffer local variable in FunctionC, the code execution goes smoothly.

```
/*  cBuffer is not properly checked for boundary */
```



```
FunctionC(parameter0’)
{
     char cBuffer[0x20]
     …
}

FunctionB(parameter0)
{
     FunctionC(parameter0);
     …
}

FunctionA(void)
{
     FunctionB(par0);
     ..
}
```

However, Figure 2 shows the cBuffer is deliberately overflowed until it reaches the first byte or least significant byte of original EBP register on stack. In other words, EBP’ becomes rounded down to (`original EBP’ & 0xFFFFFF00`). Before the FunctionB returns to calling FunctionA, the epilogue tries to restore ESP with EBP. Because EBP has been rounded down, then the stack pointer ESP is moved accordingly to point to the overflowed cBuffer area and a new return address is provided to run the malicious code. The concept of off-by-one exploitation is quite delicate and memory space limiting. The real implementation or exploitation in the wild world is unknown.

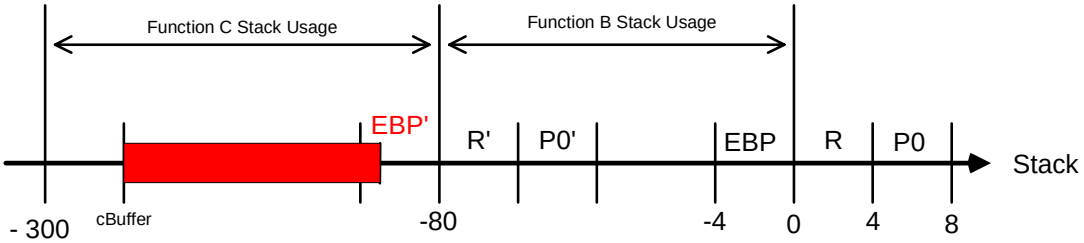

Figure 2 An Off-by-one exploitation of stack overflow with EBP' being rounded down to point to the overflowed area. Notice EBP' = EBP' & 0xFFFFFF00

One method to protect against off-by-one exploitation scheme is to utilize the debug methodology used by Visual C/C++ studio, i.e. to fill the used and to-be-discarded stack space with 0CC or 00 byte before RET in such a way as follows:

```
…
Push EAX
Push edi
Lea  edi, [esp+8]
Mov  ecx, (size of stack space used) >> 2
```



```
Mov  eax, 0ccCCccCCh     ; or 0h
Rep  stosd
Pop  edi
Pop  eax
Mov  ESP, EBP
Pop  EBP
RET
```

The purpose is to protect against the off-by-one exploitation, if it does happen, from running a malicious code and provide a chance to debug and plug this overflow software hole. Figure 3 illustrates the protection mechanism. Little performance degradation is expected for this protection mechanism.

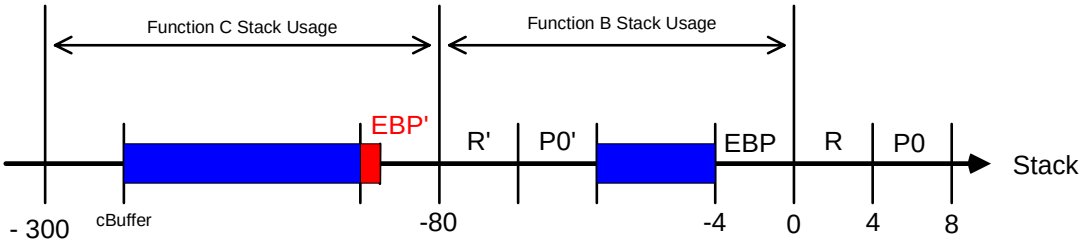

Figure 3 An Off-by-one exploitation happened. However, method of filling used stack memory space with 0CC or 00h prevents the piece of malicious code from running. Notice EBP' = EBP' & 0xFFFFFF00

3 Protection Against Heap Overflow Bugs And Exploitation

As a software engineer, you might have come across situations where memory trampling happened somehow and nobody knew why it happened. Runtime double free bugs (4) and heap overflow exploitation (3) could allow a chunk of malicious code to be executed. Figure 4 illustrates a normal heap structure with two memory blocks allocated.

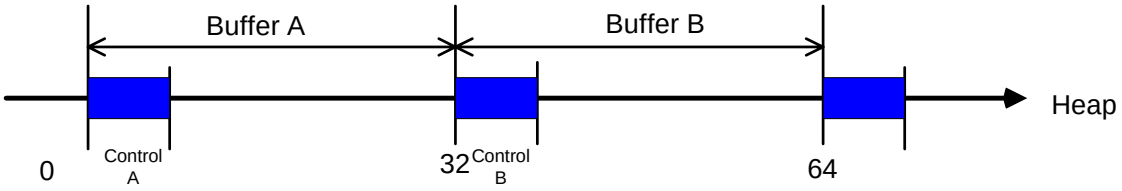

Figure 4 Heap Blocks Used by Win2k And Borland C++ Run-time Library Control Block is BLUE.

If the usage of buffer A is mal-programmed and the buffer is overflowed to the nearby buffer control B as Figure 5 illustrates, then the subsequent manipulation of buffer B such as *free* or *realloc* function calls could "allow an unauthenticated, remote attacker with read-only access to execute arbitrary code, alter program operation, read sensitive information, or cause a denial of service" (4).

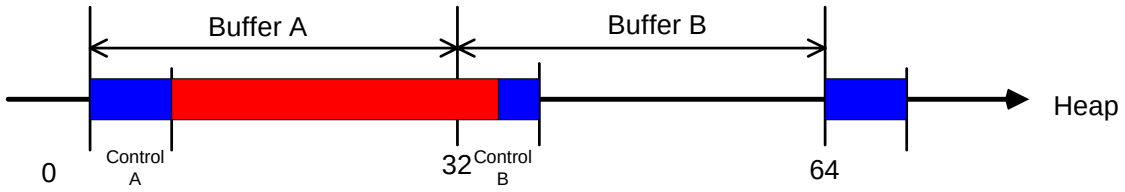

Figure 5 Buffer A is overflowed to the control structure B

Then, how do we protect against heap-based overflow exploitation? The following is an answer to solving the heap-based overflow and helping debug heap-based memory trampling during software development life cycle.

3.1 Global Initialization And Heap Control Structure

The heap control structure (32-bit CPU version) could be defined such as:

```
typedef _HEAP_CONTROL
{
    U32 anchor;     /* unsigned int 32-bit or 64-bit */
    more structure members
} HEAP_CONTROL, * PHEAP_CONTROL;
```

During the global initialization routine, the globalRandom is initialized to provide a random number.

```
U32  globalRandom = 0;

….
GlobalRandom = rand();
…
```

3.2 Allocation Of A New Heap Block

For example, the `malloc` or `calloc` function can be programmed as such:

- Check the requested size against limits;
- If requested size is allowed, then manipulate the memory control chains to get a block of memory with size as:
  `ROUND_UP_ALIGNMENT(sizeof(HEAP_CONTROL) + requested size) )`
- Let's assume the (`PHEAP_CONTROL`) pAddr point to the free memory block retrieved. The HEAP_CONTROL structure is initialized as follows. The `op` can be add, subtract, and XOR etc.
  `pAddr->anchor = ( (U32) pAddr ) op globalRandom`
- More initialization before returning pointer to the buffer after the buffer control structure.

3.3 free, realloc and integrity-check heap memory

During the manipulation of heap memory such as *free*, *realloc* or heap integrity check algorithm (either run-time or postmortem debug), the *(void *) memblock* parameter can be checked as such (32-bit version):

```
void free(void * memblock)
{
    PHEAP_CONTROL  pHeap;
     if ( (U32) memblock < sizeof(HEAP_CONTROL) )
     {
         error("invalid memblock pointer");
         SystemPanic();      /* report this problem */
                             /* so that it can be fixed */
     }

     pHeap = (PHEAP_CONTROL)
             ( (U32) memblock - sizeof(HEAP_CONTROL) );

     if ( ( (U32) pHeap op globalRandom ) !=
           pHeap->anchor )
     {
         DumpRelavantInfoForDebug();     /* dump info */
         SystemPanic();      /* report this problem */
                             /* so that it can be fixed */
     }

     pHeap->anchor = 0;

     normal free continue;

     released memory block can be zeroed out or filled with 0CC
     byte depending on how paranoid one wants to be
}
```

The purpose of the *anchor* member in the HEAP_CONTROL structure is to utilize the address structure itself to determine whether it has been overflowed. The *globalRandom* introduces a randomness to make the specific-memory-block-directed overflowing impossible. Even if the heap control structure is overflowed, it is very unlikely that it will cause any harm except one shot of DoS for this application due to this anchoring mechanism.

This anchoring technique can also be applied to a memory block, either a data structure or some other critical info, to protect against being overflowed by misuse or abuse of preceding memory blocks. Obviously, this technique is very helpful as well for software developers in chasing memory trampling or overflowing bugs, and locating double-free bugs during normal development cycles.

4 Protect Against Overflowed Exception Handling Frame Like Code Red

One method to protect against overflowed structured exception handling frame, whether stake-based or heap-based, is to include an anchor as the first member of the structured exception handling (SEH) structure. During the initialization, the *anchor* in the structure is initialized as such:

```
pStructure->anchor =
   ((U32) pStructure) op globalRandom;
```

During the exception handling, the operating system checks the integrity of the SEH structure as the section 3.3 illustrates before it transfers the rein to the SEH routine. Even if the SEH structure is overflowed as Code Red did to Microsoft IIS, the malicious code will not be executed due to the fact the malicious code cannot figure out the exact result processed from the SEH address and the system random number.

5 Refuse To RET To Defend Against Brute Force Exploitation

My previous paper addressed the exploitation mechanism used by SQL Slammer worm by inserting 0CCh opcode(s) onto the to-be-discarded stack space used normally for parameter passing to trap the "jmp/call ESP" exploits (1). However, it does not address the brute force exploitation of RET addresses seen in WebDAV exploitation. The WebDAV exploitation utilizes the unchecked boundary overflow during the Unicode conversion of abnormally long data. Some exploit codes posed on the Internet utilize a brute force return address mechanism to redirect normal execution flow to the exploit code.

Then, we can take advantage of the anchoring mechanism to halt before RET if the program detects an overflowed stack.

5.1 Code Generation Utilizes Anchoring Mechanism

For a normal function, either C or Pascal-style, the prologue would be like the following to place the *globalRandom* value just below the return address.

```
Push EBP
Push EDI
Push ESI
Push EBX
Push DWORD PTR [address of globalRandom] ; or push ESP
Mov  EBP, ESP
….
```

Then, the following epilogue code would check whether the return address is overflowed and replaced.

```
Mov  ESP, EBP
pop  ecx
cmp  [address of globalrandom], ecx; or cmp esp, ecx
```



```
     jz   goret
     db   0cch           ; single-step exception if overflowed
goret:
     pop  EBX
     pop  ESI
     pop  EDI
     pop  EBP
     ret                 ; or ret x
```

5.2 Diagram Illustration On This Mechanism

Figure 6 illustrates the placement of an anchor below the RET address along with saved registers (or below a block of data on stack or a heap control structure mentioned above) in the prologue code as a detection mechanism.

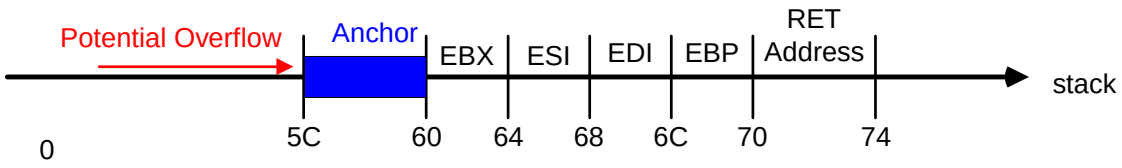

Figure 6 An anchor is placed below saved registers and the RET address to detect potential overflow.
Anchoring Data (globalRandom value) in Blue

Figure 7 illustrates the utilization of the anchor value before RET to halt the execution of code if the stack has been overflowed.

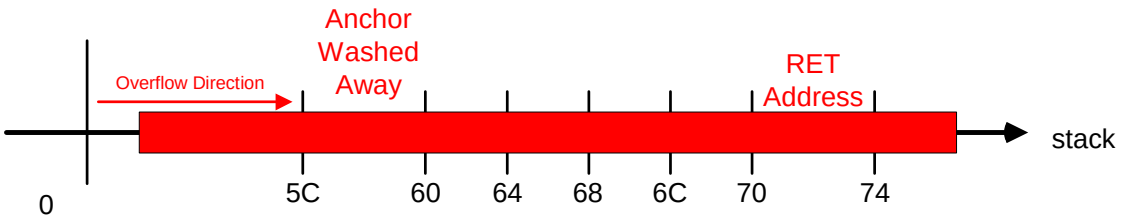

Figure 7 Code Execution Halts When Return Address is Overflowed
Overflowing data in RED

With this mechanism implemented, it is more convenient than the method published earlier (1) and will not have a library compatibility problem and have a stronger protection.

6 Conclusion

The fundamental vulnerability of overflow exploitation code lies in the fact the overflow happens just like water runs on the prairie. It does not jump! Once uniquely positioned anchors are washed away, then it is certain that overflowing happened!

The off-by-one exploitation of overflowed stack can be prevented by utilization of the debug methodology to fill the used, to-be-discarded, stack memory space with 0 or 0CC byte. With the

anchor mechanism illustrated in Figure 6 and Figure 7 implemented, it is unnecessary to fill the to-be-ignored stack with 0 or 0CC byte any more.

The heap manipulation such as *malloc* or *free* etc. functions can be improved to include an anchor member in the heap control structure to utilize the address of the structure itself to protect against the heap-overflowed exploitation or help debug memory trampling during normal software development cycle. This anchoring mechanism can be applied to other programming areas for protection against overflowing as well. However, this mechanism would not protect against address-specific overwriting such as format string bugs.

The anchoring mechanism can used to protect against the exploitation of Structured Exception Handling like Code Red to Microsoft IIS because the overflowed code cannot determine the exact result of the SEH address and the system random number.

The anchoring mechanism also enables the normal execution of code to halt on the brink of RET if it detects the return address is overflowed to defend against brute-force exploitation mechanism seen in the WebDAV and others.

For a simplified version of anchoring mechanism, the anchor has to be the first member of a structure or occupies the first few bytes of a memory block at its lowest address, and is initialized with the system random number and verified against the random number during runtime validation.

Protection mechanisms do have a little, even though minimal, impact on the runtime performance. Even if software engineers wrote solid, bug-free code, it is still beneficial to have protection mechanisms implemented against unknown situations. In other words, protection mechanism implemented is still better than a sense of security presumed, I believe.

7 Reference

1. http://www.securityfocus.com/archive/1
2. http://www.virusbtn.com/index.xml
3. http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt
4. http://www.cert.org/advisories/CA-2003-02.html
5. http://www.peterszor.com/blended.pdf
6. http://www.symantec.com/avcenter/security/Content/3.17.2003.html

About Author

Yinrong Huang was trained as a biologist and began his computer programmer career by self-studying assembly language for Intel. He has worked on real operating systems such as Windows, Unix to real-time operating system such as VxWorks and wrote device drivers for Solaris, VxWorks and Windows. He wrote Board Suport Package, boot-up firmware as well as application. He is familiar with Intel x86, PPC, Hitachi SH3, MIP and Sparc CPU architectures and CPCI, PCI, and 1394 bus architectures. His programming languages include assembly, C/C++, TCL/TK, Perl, Forth, and scripts.

If you finish reading his paper and want to offer him a research and/or programming position in your company or offer him financial funding into his research or license his self-protection and other security-related programming products and concept, you are welcome to contact him at:

Yinrong@rogers.com

Thanks for reading.
