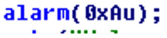
Signed Shell Server for newbies

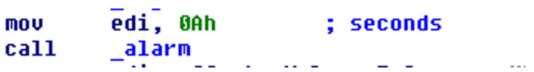
Introduction

- TODO

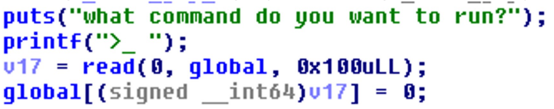
Patching the timeout alarm

- Nop the following command (using IDA) to prevent from the program from exiting after the timeout period
- 

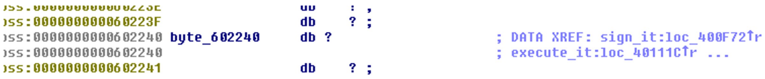
```
alarm(0xAu);
```

- 

```
mov     edi, 0Ah            ; seconds
call    _alarm
```

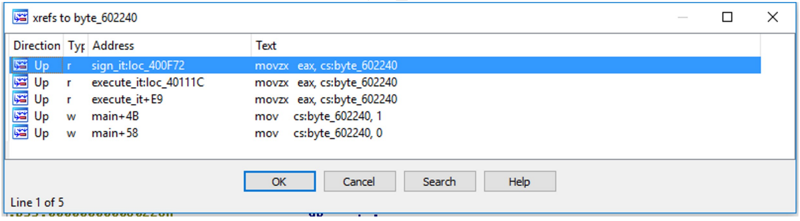
- Open the ELF file via IDA x64
- Navigate to execute_it function
- Investigate the following input handling some user input:
- 

```
puts("what command do you want to run?");
printf(">_ ");
v17 = read(0, global, 0x100uLL);
global[(signed __int64)v17] = 0;
```

- Read about off by one vulnerability : https://www.exploit-db.com/docs/28478.pdf
- v17 = the number of chars that was fed into global (max 0x100 chars)
- global = global char array with constant size - 0x100 (256 chars)
- If the user input is exactly 256 chars, then the '\0' will be written into the next place after the 'global' buffer
- Using IDA, look at the variable located after the global array (in the bss section):
-
- 

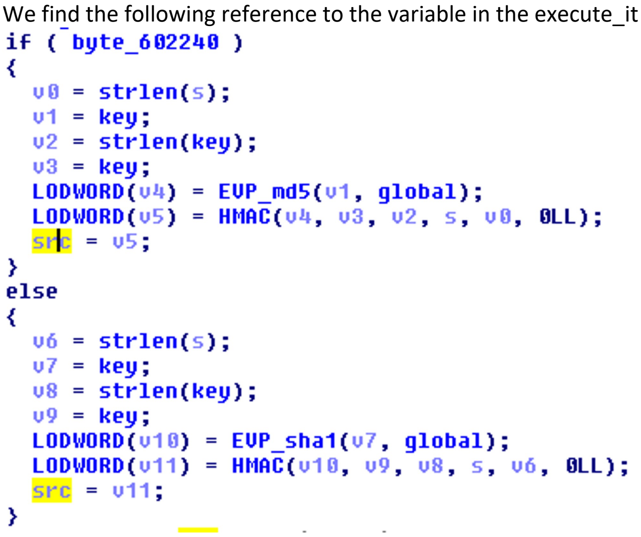
```
bss:000000000060223E                 db    ? ;
bss:000000000060223F                 db    ? ;
bss:0000000000602240 byte_602240     db ?                    ; DATA XREF: sign_it:loc_400F72↑r
bss:0000000000602240                                         ; execute_it:loc_40111C↑r ...
bss:0000000000602241                 db    ? ;
```

- Using the off by one vulnerability, we can set the var byte_602240 to 0
- Now let's investigate how will it affect the flow of the program

- Click 'X' to see all the references to byte_602240
- 

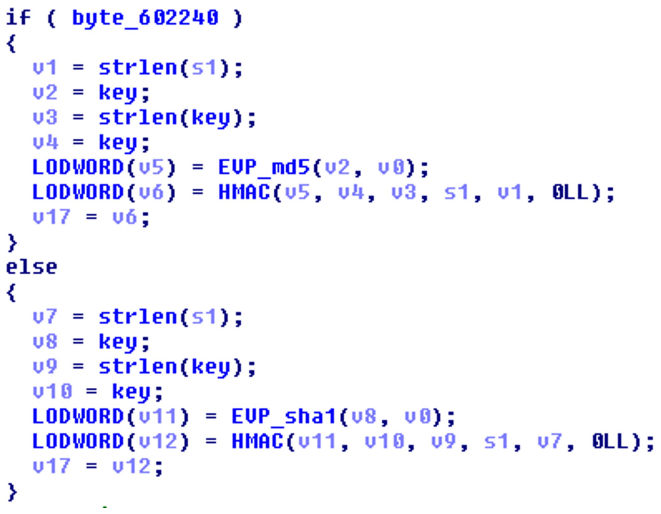
xrefs to byte_602240

Direction	Typ	Address	Text
Up	r	sign_it:loc_400F72	movzx eax, cs:byte_602240
Up	r	execute_it:loc_40111C	movzx eax, cs:byte_602240
Up	r	execute_it+E9	movzx eax, cs:byte_602240
Up	w	main+4B	mov cs:byte_602240, 1
Up	w	main+58	mov cs:byte_602240, 0

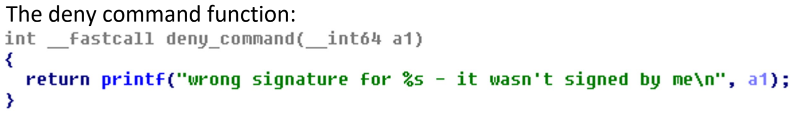
- We find the following reference to the variable in the execute_it func:
- 

```
if ( byte_602240 )
{
  v0 = strlen(s);
  v1 = key;
  v2 = strlen(key);
  v3 = key;
  LODWORD(v4) = EVP_md5(v1, global);
  LODWORD(v5) = HMAC(v4, v3, v2, s, v0, 0LL);
  src = v5;
}
else
{
  v6 = strlen(s);
  v7 = key;
  v8 = strlen(key);
  v9 = key;
  LODWORD(v10) = EVP_sha1(v7, global);
  LODWORD(v11) = HMAC(v10, v9, v8, s, v6, 0LL);
  src = v11;
}
```

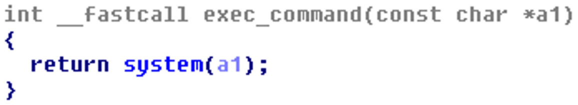
- And a similar reference in the sign_it func:

- 

```
if ( byte_602240 )
{
  v1 = strlen(s1);
  v2 = key;
  v3 = strlen(key);
  v4 = key;
  LODWORD(v5) = EVP_md5(v2, v0);
  LODWORD(v6) = HMAC(v5, v4, v3, s1, v1, 0LL);
  v17 = v6;
}
else
{
  v7 = strlen(s1);
  v8 = key;
  v9 = strlen(key);
  v10 = key;
  LODWORD(v11) = EVP_sha1(v8, v0);
  LODWORD(v12) = HMAC(v11, v10, v9, s1, v7, 0LL);
  v17 = v12;
}
```

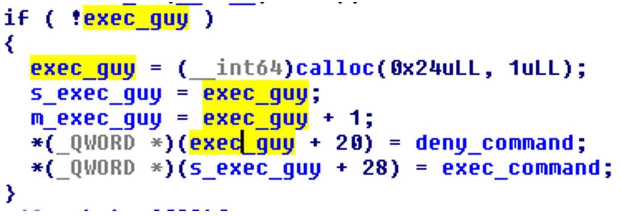
- Now let's examine the code flow and find out a way to use this var
- The deny command function:
- 

```
int __fastcall deny_command(__int64 a1)
{
  return printf("wrong signature for %s - it wasn't signed by me\n", a1);
}
```

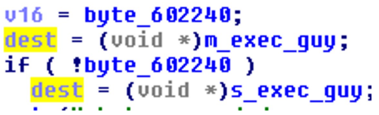
- Not running the command due to a bad signature, printing an error
- The exec_command func:
- 

```
int __fastcall exec_command(const char *a1)
{
  return system(a1);
}
```

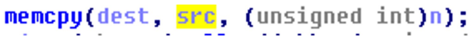
- Running the command using "system"

- Examine the execute_it function flow
- The exec_guy struct
- 

```
if ( !exec_guy )
{
  exec_guy = (__int64)calloc(0x24uLL, 1uLL);
  s_exec_guy = exec_guy;
  m_exec_guy = exec_guy + 1;
  *(_QWORD *)(exec_guy + 20) = deny_command;
  *(_QWORD *)(s_exec_guy + 28) = exec_command;
}
```

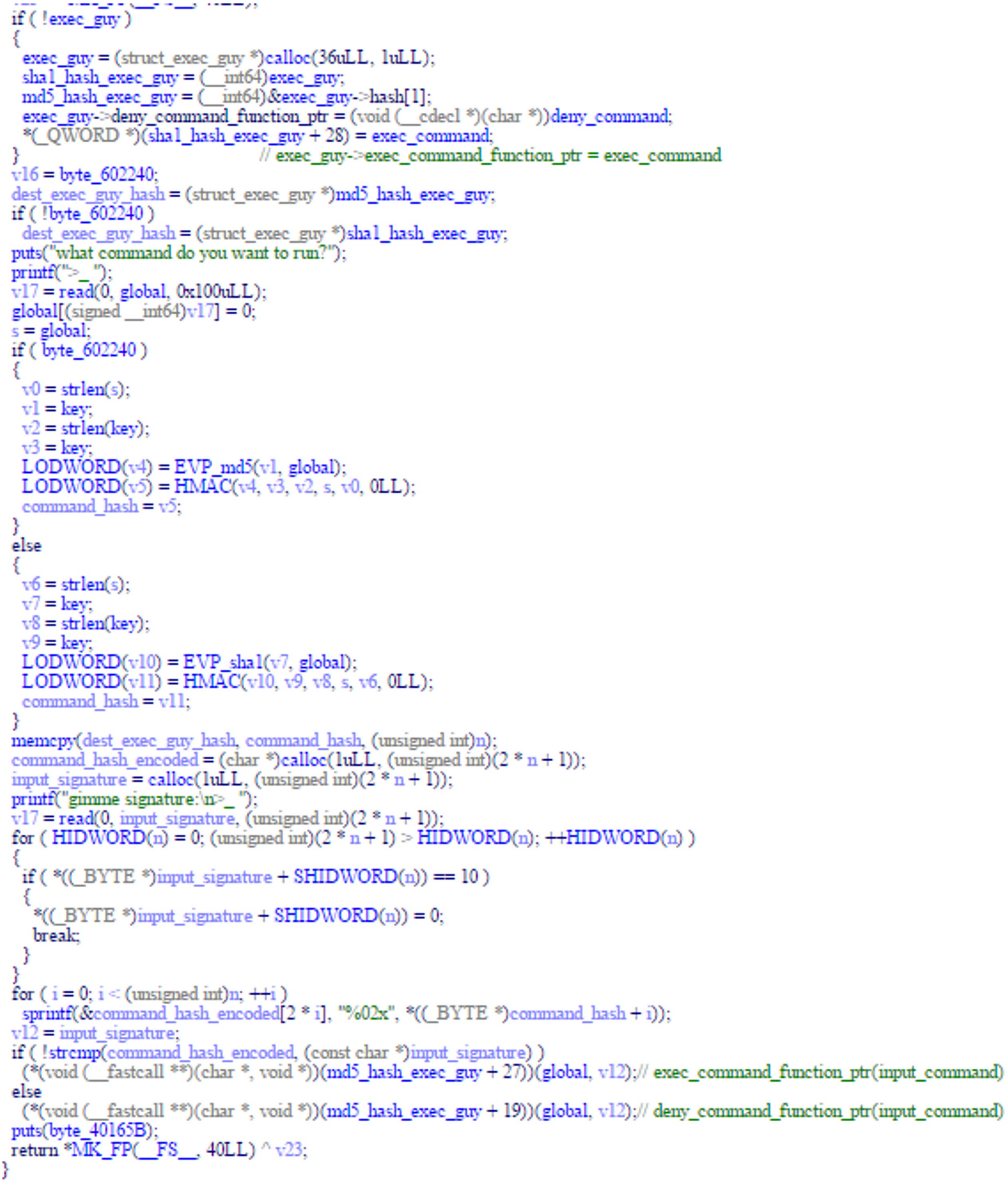
- The exec_guy variable is a struct 0x24 size long contains the following fields:
  - Hash who is array of bytes field, the first field in the struct (starting at offset 0 in the struct) that will contain the hash of the input command (md5 or sha1)
  - A pointer to the deny_command function, starting at offset 20 in the struct
  - A pointer to the exec_command function, starting at offset 28 in the struct
  - s_exec_guy is a pointer to the hash field in the exec_guy struct, pointing to the start of the hash array field
  - m_exec_guy is a pointer to the hash field in the exec_guy struct, pointing to one byte after the start of the hash array field
- 

```
v16 = byte_602240;
dest = (void *)m_exec_guy;
if ( !byte_602240 )
  dest = (void *)s_exec_guy;
```

- 

```
memcpy(dest, src, (unsigned int)n);
```

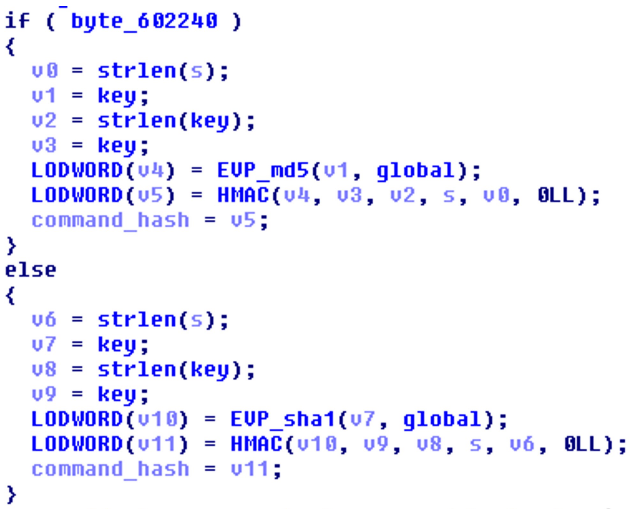
- Now let's add some meaningful names to the variables and add comments describing the flow of the function



```
  if ( !exec_guy )
  {
    exec_guy = (struct_exec_guy *)calloc(36uLL, 1uLL);
    sha1_hash_exec_guy = (__int64)exec_guy;
    md5_hash_exec_guy = (__int64)&exec_guy->hash[1];
    exec_guy->deny_command_function_ptr = (void (__cdecl *)(char *))deny_command;
    *(_QWORD *)(sha1_hash_exec_guy + 28) = exec_command;
  }                                             // exec_guy->exec_command_function_ptr = exec_command
  v16 = byte_602240;
  dest_exec_guy_hash = (struct_exec_guy *)md5_hash_exec_guy;
  if ( !byte_602240 )
    dest_exec_guy_hash = (struct_exec_guy *)sha1_hash_exec_guy;
  puts("what command do you want to run?");
  printf(">_ ");
  v17 = read(0, global, 0x100uLL);
  global[(signed __int64)v17] = 0;
  s = global;
  if ( byte_602240 )
  {
    v0 = strlen(s);
    v1 = key;
    v2 = strlen(key);
    v3 = key;
    LODWORD(v4) = EVP_md5(v1, global);
    LODWORD(v5) = HMAC(v4, v3, v2, s, v0, 0LL);
    command_hash = v5;
  }
  else
  {
    v6 = strlen(s);
    v7 = key;
    v8 = strlen(key);
    v9 = key;
    LODWORD(v10) = EVP_sha1(v7, global);
    LODWORD(v11) = HMAC(v10, v9, v8, s, v6, 0LL);
    command_hash = v11;
  }
  memcpy(dest_exec_guy_hash, command_hash, (unsigned int)n);
  command_hash_encoded = (char *)calloc(1uLL, (unsigned int)(2 * n + 1));
  input_signature = calloc(1uLL, (unsigned int)(2 * n + 1));
  printf("gimme signature:\n>_ ");
  v17 = read(0, input_signature, (unsigned int)(2 * n + 1));
  for ( HIDWORD(n) = 0; (unsigned int)(2 * n + 1) > HIDWORD(n); ++HIDWORD(n) )
  {
    if ( *((_BYTE *)input_signature + SHIDWORD(n)) == 10 )
    {
      *((_BYTE *)input_signature + SHIDWORD(n)) = 0;
      break;
    }
  }
  for ( i = 0; i < (unsigned int)n; ++i )
    sprintf(&command_hash_encoded[2 * i], "%02x", *((_BYTE *)command_hash + i));
  v12 = input_signature;
  if ( !strcmp(command_hash_encoded, (const char *)input_signature) )
    (*(void (__fastcall **)(char *, void *))(md5_hash_exec_guy + 27))(global, v12);// exec_command_function_ptr(input_command)
  else
    (*(void (__fastcall **)(char *, void *))(md5_hash_exec_guy + 19))(global, v12);// deny_command_function_ptr(input_command)
  puts(byte_40165B);
  return *MK_FP(__FS__, 40LL) ^ v23;
}
```

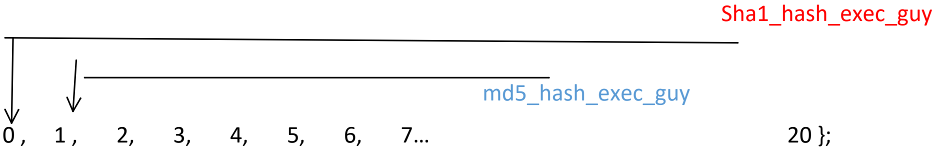
- execute_it

- The var byte_602240 is the one setting the function flow to hash the command in sha-1 or in md5
- If byte_602240 == 1 :
    - Hash the command using md5
    - dest_exec_guy_hash = md5_hash_exec_guy = exec_guy + 1
- If byte_602240 == 0 :
    - Hash the command using sh1
    - dest_exec_guy_hash = sha1_hash_exec_guy = exec_guy (+0 offset)
- Pay attention that dest_exec_guy_hash point is being set only at the first loop of the problem (to point to exec_guy or exec_guy + 1 offset)
- But, the decision to hash the command using sha1 or using md5 is checked on each run (by byte_602240):

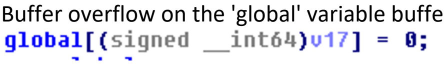


```
if ( byte_602240 )
{
  v0 = strlen(s);
  v1 = key;
  v2 = strlen(key);
  v3 = key;
  LODWORD(v4) = EVP_md5(v1, global);
  LODWORD(v5) = HMAC(v4, v3, v2, s, v0, 0LL);
  command_hash = v5;
}
else
{
  v6 = strlen(s);
  v7 = key;
  v8 = strlen(key);
  v9 = key;
  LODWORD(v10) = EVP_sha1(v7, global);
  LODWORD(v11) = HMAC(v10, v9, v8, s, v6, 0LL);
  command_hash = v11;
}
```

- The reason there is a difference in the offset in the hash array field between the sha1 and the md5 is that the hash size of sha1 is bigger then the hash size of md5
- struct exec_guy

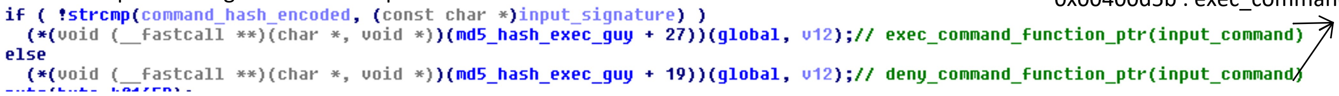


```
{
 char hash[20] = { 0 ,   1 ,    2,     3,     4,     5,     6,     7…                                   20 };
   void (*deny_command_function_ptr)(char*);
   void (*exec_command_function_ptr)(char*);
}
```

- Exploit flow
  - byte_602240 = 1
  - dest_exec_guy_hash = md5_hash_exec_guy = exec_guy + 1
  - Enter 256 long input
  - Buffer overflow on the 'global' variable buffer => the var after it is set to 0 :
  - 

    ```
    global[(signed __int64)v17] = 0;
    ```
  - global[256] = global[0] + 256 = byte_602240 = 0
  - In the next run of the execute_it func :
  - dest_exec_guy_hash = md5_hash_exec_guy = exec_guy + 1 (still)
  - byte_602240 = 0
  - The command will be hashed using sha1
  - The sha1 hash will be stored in the dest_exec_guy_hash = md5_hash_exec_guy = exec_guy + 1
  - Since the sha1 hash is 20 bytes long, the LSB of the next field after the hash field in the exec_guy struct will be overwritten with the last byte of the hash
  - The next field after the hash is the deny_command_function_ptr
  - The target is to overwrite this byte so that the pointer to the deny_command function, will actually point to the exec_command function
  - deny_command_function_ptr = 0x00400d36
  - exec_command_function_ptr = 0x00400d5b
  - So, the goal is to find a command such that the last byte of the SHA-1 digest is 0x5b
- After the exploit, the deny_command_function_ptr will point to the exec_command function, so our command will be executed even when the input hash signature is not equal to the command hash

0x00400d5b : exec_command



```
if ( !strcmp(command_hash_encoded, (const char *)input_signature) )
  (*(void (__fastcall **)(char *, void *))(md5_hash_exec_guy + 27))(global, v12);// exec_command_function_ptr(input_command)
else
  (*(void (__fastcall **)(char *, void *))(md5_hash_exec_guy + 19))(global, v12);// deny_command_function_ptr(input_command)
```