A walk through the binary with IDA

January 28, 2016

Want to learn reverse engineering? Continue ahead. This is a walkthrough of the "keygenme" reversing challenge from NYU's CSAW 2013 CTF competition. I try to be thorough and highlight how to solve both a CTF challenge, and how to use standard and modern reverse engineering tools.

> Reversing : keygenme
>
> someone has leaked a binary from an activation server.
> can you crack the keygen algorithm for me?
>
> using the ELF provided, reverse the keygeneration algorithm.
> The server listening at raxcity.com on port 2000 will ask you for
> the passwords of various usernames. If you can provide 10 passwords, you might get a nice flag :-)
>
> *hint*
> Rumor has it that the actual keygen runs in a custom vm. I'd start by decoding the instruction format.

Try to make it break

Play with the binary and give it inputs. See if you can draw conclusions, or make it crash.

```
$ ./keygenme32.elf
usage: ./keygenme32.elf <username> <token 1> <token 2>
```

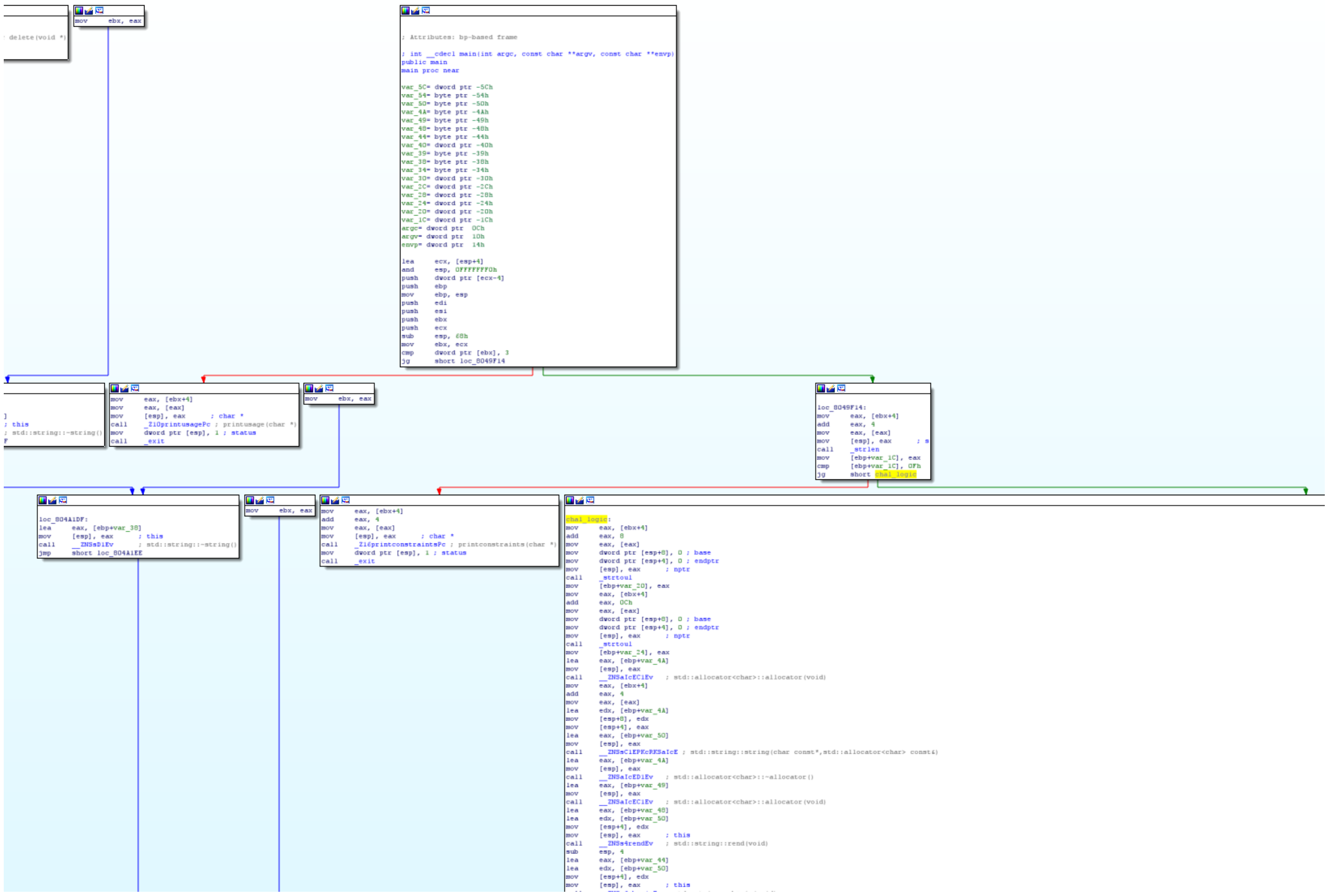
```
$ ./keygenme32.elf hellohello 123 123
error: hellohello is not a valid name
```

```
$ ./keygenme32.elf hellohellohello1 123 123
:-(
```

Simply running it gives us the usage details. Trying out different sized usernames leads us to figuring out that the username for the keygen has to be 16+ characters. I couldn't get any crashes, which is reasonable since it's a "reversing" challenge, and not a pwning challenge.

High level picture from disassembly

Let's use IDA to see what the binary is about.



```
mov     ebx, eax
```

```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_5C= dword ptr -5Ch
var_54= byte ptr -54h
var_50= byte ptr -50h
var_4A= byte ptr -4Ah
var_49= byte ptr -49h
var_48= byte ptr -48h
var_44= byte ptr -44h
var_40= dword ptr -40h
var_39= byte ptr -39h
var_38= byte ptr -38h
var_34= byte ptr -34h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
argc= dword ptr  0Ch
argv= dword ptr  10h
envp= dword ptr  14h

lea     ecx, [esp+4]
and     esp, 0FFFFFFF0h
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    edi
push    esi
push    ebx
push    ecx
sub     esp, 68h
mov     ebx, ecx
cmp     dword ptr [ebx], 3
jg      short loc_8049F14
```

```
mov     eax, [ebx+4]
mov     eax, [eax]
mov     [esp], eax       ; char *
call    _Z10printusagePc ; printusage(char *)
mov     dword ptr [esp], 1 ; status
call    _exit
```

```
mov     ebx, eax
```

```
loc_8049F14:
mov     eax, [ebx+4]
add     eax, 4
mov     eax, [eax]
mov     [esp], eax       ; s
call    _strlen
mov     [ebp+var_1C], eax
cmp     [ebp+var_1C], 0Fh
jg      short chal_logic
```

```
loc_804A1DF:
lea     eax, [ebp+var_38]
mov     [esp], eax       ; this
call    __ZNSsD1Ev       ; std::string::~string()
jmp     short loc_804A1EE
```

```
mov     ebx, eax
```

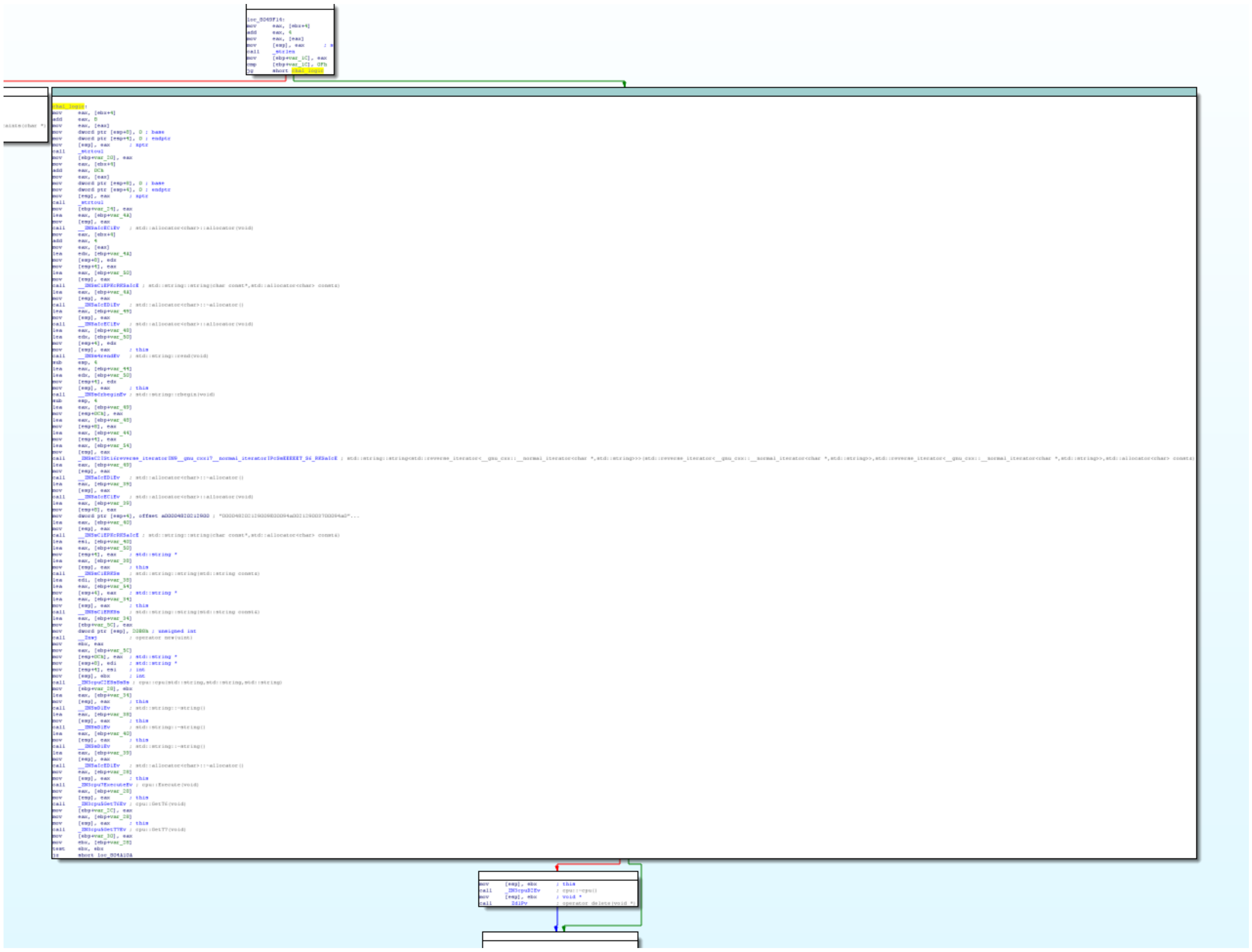
```
mov     eax, [ebx+4]
add     eax, 4
mov     eax, [eax]
mov     [esp], eax       ; char *
call    _Z16printconstraintsPc ; printconstraints(char *)
mov     dword ptr [esp], 1 ; status
call    _exit
```

```
chal_logic:
mov     eax, [ebx+4]
add     eax, 8
mov     eax, [eax]
mov     dword ptr [esp+8], 0 ; base
mov     dword ptr [esp+4], 0 ; endptr
mov     [esp], eax       ; nptr
call    _strtoul
mov     [ebp+var_20], eax
mov     eax, [ebx+4]
add     eax, 0Ch
mov     eax, [eax]
mov     dword ptr [esp+8], 0 ; base
mov     dword ptr [esp+4], 0 ; endptr
mov     [esp], eax       ; nptr
call    _strtoul
mov     [ebp+var_24], eax
lea     eax, [ebp+var_4A]
mov     [esp], eax
call    __ZNSaIcEC1Ev    ; std::allocator<char>::allocator(void)
mov     eax, [ebx+4]
add     eax, 4
mov     eax, [eax]
lea     edx, [ebp+var_4A]
mov     [esp+8], edx
mov     [esp+4], eax
lea     eax, [ebp+var_50]
mov     [esp], eax
call    __ZNSsC1EPKcRKSaIcE ; std::string::string(char const*,std::allocator<char> const&)
lea     eax, [ebp+var_4A]
mov     [esp], eax
call    __ZNSaIcED1Ev    ; std::allocator<char>::~allocator()
lea     eax, [ebp+var_49]
mov     [esp], eax
call    __ZNSaIcEC1Ev    ; std::allocator<char>::allocator(void)
lea     eax, [ebp+var_48]
lea     edx, [ebp+var_50]
mov     [esp+4], edx
mov     [esp], eax       ; this
call    __ZNSs4rendEv    ; std::string::rend(void)
sub     esp, 4
lea     eax, [ebp+var_44]
lea     edx, [ebp+var_50]
mov     [esp+4], edx
mov     [esp], eax       ; this
```

We enter *main*. The first fork leads us to either calling `printusage()` or not.
We probably want to go down the path of not calling `printusage()`.

Next fork is the length check that we already figured out.

And then bam! A gigantic linear procedure.
This looks like a lot of work. Trying to decompile with the *tab* hotkey doesn't work. We can try to decompile the entire thing (*Ctrl+F5*), and see if that's any better -- nope, the `main()` didn't decompile for some reason.



```
loc_8049F14:
mov     eax, [ebx+4]
add     eax, 4
mov     eax, [eax]
mov     [esp], eax      ; s
call    _strlen
mov     [ebp+var_1C], eax
cmp     [ebp+var_1C], 0Fh
jg      short chal_logic
```

```
chal_logic:
mov     eax, [ebx+4]
add     eax, 8
mov     eax, [eax]
mov     dword ptr [esp+8], 0 ; base
mov     dword ptr [esp+4], 0 ; endptr
mov     [esp], eax      ; nptr
call    _strtoul
mov     [ebp+var_20], eax
mov     eax, [ebx+4]
add     eax, 0Ch
mov     eax, [eax]
mov     dword ptr [esp+8], 0 ; base
mov     dword ptr [esp+4], 0 ; endptr
mov     [esp], eax      ; nptr
call    _strtoul
mov     [ebp+var_24], eax
lea     eax, [ebp+var_4A]
mov     [esp], eax
call    __ZNSaIcEC1Ev   ; std::allocator<char>::allocator(void)
mov     eax, [ebx+4]
add     eax, 4
mov     eax, [eax]
lea     edx, [ebp+var_4A]
mov     [esp+8], edx
mov     [esp+4], eax
lea     eax, [ebp+var_50]
mov     [esp], eax
call    __ZNSsC1EPKcRKSaIcE ; std::string::string(char const*,std::allocator<char> const&)
lea     eax, [ebp+var_4A]
mov     [esp], eax
call    __ZNSaIcED1Ev   ; std::allocator<char>::~allocator()
lea     eax, [ebp+var_49]
mov     [esp], eax
call    __ZNSaIcEC1Ev   ; std::allocator<char>::allocator(void)
lea     eax, [ebp+var_48]
lea     edx, [ebp+var_50]
mov     [esp+4], edx
mov     [esp], eax      ; this
call    __ZNSs4rendEv   ; std::string::rend(void)
sub     esp, 4
lea     eax, [ebp+var_44]
lea     edx, [ebp+var_50]
mov     [esp+4], edx
mov     [esp], eax      ; this
call    __ZNSs6rbeginEv ; std::string::rbegin(void)
sub     esp, 4
lea     eax, [ebp+var_49]
mov     [esp+0Ch], eax
lea     eax, [ebp+var_48]
mov     [esp+8], eax
lea     eax, [ebp+var_44]
mov     [esp+4], eax
lea     eax, [ebp+var_54]
mov     [esp], eax
call    _ZNSsC2ISt16reverse_iteratorIN9__gnu_cxx17__normal_iteratorIPcSsEEEEET_S6_RKSaIcE ; std::string::string<std::reverse_iterator<__gnu_cxx::__normal_iterator<char *,std::string>>>(std::reverse_iterator<__gnu_cxx::__normal_iterator<char *,std::string>>,std::reverse_iterator<__gnu_cxx::__normal_iterator<char *,std::string>>,std::allocator<char> const&)
lea     eax, [ebp+var_49]
mov     [esp], eax
call    __ZNSaIcED1Ev   ; std::allocator<char>::~allocator()
lea     eax, [ebp+var_39]
mov     [esp], eax
call    __ZNSaIcEC1Ev   ; std::allocator<char>::allocator(void)
lea     eax, [ebp+var_39]
mov     [esp+8], eax
mov     dword ptr [esp+4], offset a00004820212900 ; "00004820212900980094a0021290037000094a0"...
lea     eax, [ebp+var_40]
mov     [esp], eax
call    __ZNSsC1EPKcRKSaIcE ; std::string::string(char const*,std::allocator<char> const&)
lea     esi, [ebp+var_40]
lea     eax, [ebp+var_50]
mov     [esp+4], eax    ; std::string *
lea     eax, [ebp+var_38]
mov     [esp], eax      ; this
call    __ZNSsC1ERKSs   ; std::string::string(std::string const&)
lea     edi, [ebp+var_38]
lea     eax, [ebp+var_54]
mov     [esp+4], eax    ; std::string *
lea     eax, [ebp+var_34]
mov     [esp], eax      ; this
call    __ZNSsC1ERKSs   ; std::string::string(std::string const&)
lea     eax, [ebp+var_34]
mov     [ebp+var_5C], eax
mov     dword ptr [esp], 2080h ; unsigned int
call    __Znwj          ; operator new(uint)
mov     ebx, eax
mov     eax, [ebp+var_5C]
mov     [esp+0Ch], eax  ; std::string *
mov     [esp+8], edi    ; std::string *
mov     [esp+4], esi    ; int
mov     [esp], ebx      ; int
call    _ZN3cpuC2ESsSsSs ; cpu::cpu(std::string,std::string,std::string)
mov     [ebp+var_28], ebx
lea     eax, [ebp+var_34]
mov     [esp], eax      ; this
call    __ZNSsD1Ev      ; std::string::~string()
lea     eax, [ebp+var_38]
mov     [esp], eax      ; this
call    __ZNSsD1Ev      ; std::string::~string()
lea     eax, [ebp+var_40]
mov     [esp], eax      ; this
call    __ZNSsD1Ev      ; std::string::~string()
lea     eax, [ebp+var_39]
mov     [esp], eax
call    __ZNSaIcED1Ev   ; std::allocator<char>::~allocator()
mov     eax, [ebp+var_28]
mov     [esp], eax      ; this
call    _ZN3cpu7ExecuteEv ; cpu::Execute(void)
mov     eax, [ebp+var_28]
mov     [esp], eax      ; this
call    _ZN3cpu5GetT6Ev ; cpu::GetT6(void)
mov     [ebp+var_2C], eax
mov     eax, [ebp+var_28]
mov     [esp], eax      ; this
call    _ZN3cpu5GetT7Ev ; cpu::GetT7(void)
mov     [ebp+var_30], eax
mov     ebx, [ebp+var_28]
test    ebx, ebx
jz      short loc_804A1DA
```

```
mov     [esp], ebx      ; this
call    __ZN3cpuD2Ev    ; cpu::~cpu()
mov     [esp], ebx      ; void *
call    __ZdlPv         ; operator delete(void *)
```

A first pass high-level human decompilation of the huge function, that I've renamed `chal_logic()`, looks like it's along the lines of:

```
call strtoul(argv[2])
call strtoul(argv[2])
allocate some std::strings...
get a random huge constant...
    dword ptr [esp+4], offset a00004820212900...
some more string stuff...
construct a new cpu::cpu object... # the hint did say the keygen algo runs in a VM
    calls cpu::fillmemory()
cpu::execute() is called # the fake VM CPU must take that previous constant and treat it as a series of instructions by loading it into its *fake* memory
call getT6()
call getT7()
call check(int, int, int, int)
```

```
either a sadface or happyface is printed
```

You might've noticed that after the block is another branch. That branch appears to just be some C++ garbage collection, as they both merge to the same point afterwards. It can be ignored.

Digging deeper

If you paid attention, I mentioned a couple paragraphs back that you can use *tab* to switch to pseudocode. That's huge and nifty. But, IDA has a lot of other small things that makes it useful. If you take a look at the disassembly text that IDA gave us in the `chal_logic` block, you'll notice a lot of pseudo-variables.

> NOTE: Assembly comments beginning with `<---` are my own, and others are autogenerated by IDA.

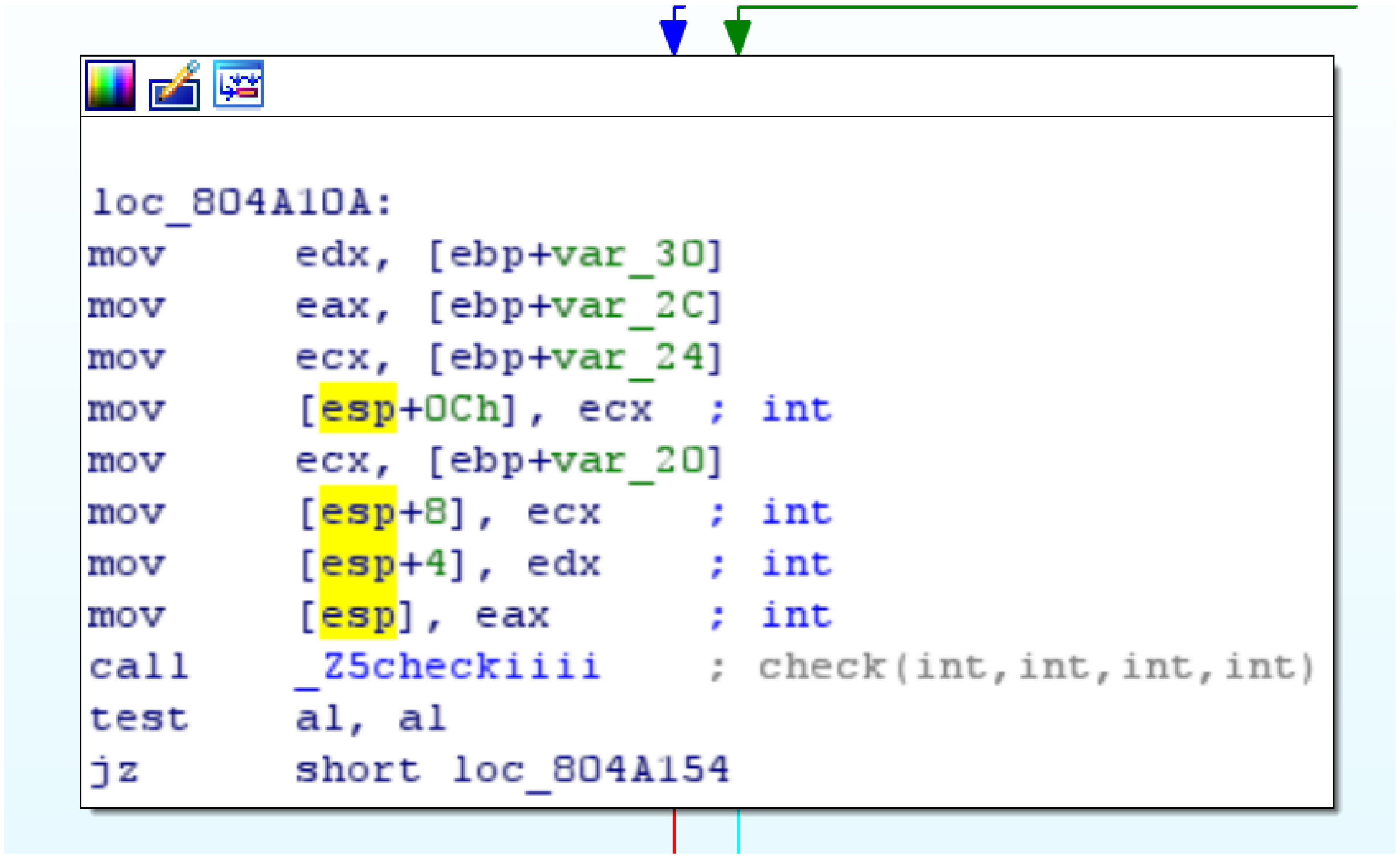
`var_28` seems to be C++'s `this` pointer for the `cpu::cpu` object. You can see this, because the `ebx` register is typically used to store the `this` pointer by C++ compilers.

```
call    _ZN3cpuC2ESsSsSs   ; cpu::cpu(std::string,std::string,std::string)
mov     [ebp+var_28], ebx  ; <--- Notice ebx, going into var_28
```

`var_2C`, and `var_30` appear to be the return values of `GetT6()` and `GetT7()`. This is evident with the simple knowledge that in x86, compilers use `eax` to store the return value of functions.

```
call    _ZN3cpu5GetT6Ev    ; cpu::GetT6(void)
mov     [ebp+var_2C], eax  ; <--- The return value of GetT6 is placed into
eax, and var_2C
mov     eax, [ebp+var_28]  ; <--- Recall that var_28 is 'this' in respect to
the cpu::cpu object
mov     [esp], eax         ; this
call    _ZN3cpu5GetT7Ev    ; cpu::GetT7(void)
mov     [ebp+var_30], eax  ; <--- The return value of GetT7 is placed into
eax, and var_30
mov     ebx, [ebp+var_28]
test    ebx, ebx           ; <--- This part is some C++ garbage collection
cruft
jz      short loc_804A10A
```

After `GetT6()` and `GetT7()` are called, a call to `check(int, int, int, int)` is done. Let's see what four ints get passed to `check()`.


```
loc_804A10A:
mov     edx, [ebp+var_30]
mov     eax, [ebp+var_2C]
mov     ecx, [ebp+var_24]
mov     [esp+0Ch], ecx  ; int
mov     ecx, [ebp+var_20]
mov     [esp+8], ecx    ; int
mov     [esp+4], edx    ; int
mov     [esp], eax      ; int
call    _Z5checkiiii    ; check(int,int,int,int)
test    al, al
jz      short loc_804A154
```

In the most common calling convention found on x86, *cdecl*, arguments get passed to functions off of the stack. In *cdecl*, the arguments of a function are pushed onto the stack in reverse order -- the function being called expects the top of the stack to be the the first argument to it. It looks like the call to `check` is `check(var_2C, var_30, var_20, var_24)`.

> NOTE: If you're not familiar with the stack, I'm going to be doing a write-up on it soon. For now, it's just a place in memory that acts like a stack data structure (last in, first out) and is used in x86 computer architecture.

From earlier we know that `var_2C` and `var_30` are `T6` and `T7` respectively. So, what is `var_20` and `var_24`? Let's look back at what IDA disassembled for us. Use the *x* hotkey on `var_20` to get xrefs to it. This will list everywhere that `var_20` is cross referenced.

```
call    _strtoul
mov     [ebp+var_20], eax  ; <--- var_20 is the return of strtoul
```

```
call    _strtoul
mov     [ebp+var_24], eax  ; <--- var_24 is the return of strtoul
```

`var_20` is the return value of the 1st `strtoul()`. And our other friend, `var_24` is the return value of the 2nd `strtoul()`.

So this `check(var_2C, var_30, var_20, var_24)` call is actually:
`check(T6 from the fake computer, T7 from the fake computer, first int input, second int input)`

Figuring out what `check()` does is actually really easy. You can slowly reverse it by using an [x86 ref manual](), or testing if IDA can do the work for us. Luckily the *tab* hotkey to see pseudo-code works.

```
_BOOL4 __cdecl check(int a1, int a2, int a3, int a4)
{
  return __PAIR__(a2, a1) == __PAIR__(
                               (unsigned __int8)a4 | (BYTE3(a4) << 8) |
((unsigned __int8)((unsigned __int16)(a4 & 0xFF00) >> 8) << 16) | ((unsigned
int)(unsigned __int8)((a4 & 0xFF0000) >> 16) << 24),
                               a3 ^ 0x31333337u);
}
```

Rewritten into python, and replacing the variable names with variable names to our liking:

```
def check(T6, T7, intarg1, intarg2):
    temp1 = intarg1 ^ 0x31333337u
    temp2 = BYTE0(intarg2) | (BYTE3(intarg2) << 8) | (BYTE1(intarg2) << 16)
| (BYTE2(intarg2) << 24)
    if temp1 != T6 or temp2 != T7:
       return false
    return true
```

The weird `__PAIR__` keyword can basically be ignored. It's just comparing the 1st arguments of each pair, and then the 2nd arguments of each pair. If both comparisons is true, then `true` is returned. The `BYTEn(x)` macro is used to mean "the nth byte of x", where `BYTE0` is the least significant byte

Okay, we almost have this binary figured out. However, we still need to know how the first argument, the *username*, is being used. It probably affects what "T6" and "T7" end up being by altering the instructions that get executed on the fake computer.

If `var_20` is the the first numeric input, `argv[2]`, then it would stand to reason that `argv[1]`, the username, is `var_1C`.

```
mov     eax, [ebx+4]
add     eax, 4
mov     eax, [eax]
mov     [esp], eax
```

```
call    _strlen
mov     [ebp+var_1C], eax
cmp     [ebp+var_1C], 0Fh  ; <--- Compare var_1C to 16
jg      short loc_8049F49
```

It looks like `var_1C` is compared to 16, so it's actually probably the length of the *username*, and not the actual username string. Going up a bit from the `cmp` instruction, it looks like the actual *username* string is in:

```
mov     eax, [ebx+4]
add     eax, 4  ; <--- eax now points to the username string
```

We can quickly leverage IDA's auto highlighting of selected variables by clicking on 'ebx' and seeing what glows yellow. The only place I see that's equiv to "ebx+8" is here:

```
mov     eax, [ebx+4]
add     eax, 4  ; <--- eax now points to the username string
mov     eax, [eax]
lea     edx, [ebp+var_4A]
mov     [esp+8], edx
mov     [esp+4], eax
lea     eax, [ebp+var_50]
mov     [esp], eax
call    __ZNSsC1EPKcRKSaIcE ; std::string::string(char
const*,std::allocator<char> const&)
```

So it would appear that the *username* is used to create a std::string. Bunch more of std::string family function calls, and we can see that the *username* string is being concatenated to the random constant string we saw earlier.

What we know so far

So far our analysis of the binary tells us:

- The binary takes 3 arguments
  - Username, a 16+ char string
  - Two integer arguments (they get converted from c-strings to ints with `strtoul()`)

- The binary constructs a `cpu::cpu`, a fake virtual computer
- The binary fills the `cpu::cpu`, with some data from a long string, "000048202129009.."

- The data is also derived from the username argument
- The binary then runs `cpu::execute` to simulate running a computer with the memory/instructions loaded into it
- The binary then gets `T6` and `T7` from that fake CPU after execution is finished.
- The binary then runs a `check(T6, T7, intarg1, intarg2)`
- If the check passes, we get a smileyface.

We know the `username` argument to the keygen is used as a seed to a fake computer to get `T6` and `T7` values out. We can reverse engineer the fake computer, and figure out how the `username` affects the instructions ran on it. However, I have a smarter and lazier idea -- it's a a computer, a fake one, but still a computer. That means it's a deterministic finite state machine. The same string input we give to it, will always produce the same output.

Recall the task at hand. We are to provide the correct key for 10 different usernames to a remote server. We have a leaked keygen binary, that takes a username and a key and tells us if they match. We can leverage this leaked keygen binary, and simply feed it the usernames that the remote server asks for. We can insert a debug break point right before the `check()` call in the binary, and see what arguments to it are. The first two arguments will be the fake computer's `T6`, and `T7`. Using those values, we can simply do math for the key that the remote server's `check()` function is expecting in return. Send it, and get a smileyface :-).

Writing our crack

We can leverage `gdb`'s python scripting capability. Note however, that the python script has to be inside of `gdb`, after you launch it. We can use my shoe.py script to talk remotely to the server. Recall also that we have to do the `check()` function in reverse. We will be getting the `T7`, and we need to figure out what we can give for `intarg` that will shuffles around and form `T7`.

```
intarg    |  a  |  b  |  c  |  d  |
               \   /     |     |
                 \      /      |
                /  \   /       |
              /      \/        |
             /      /  \       |
            v      v    v      |
T7        |  1  |  2  |  3  |  4  |
intarg =  |  3  |  1  |  2  |  4  |
```

```
import gdb
import sys
```

```
sys.path.append('.') # This is a hack to be able to import a local library
import shoe

def get_t6t7(username):
    keypart1 = "0x123123" # whatever
    keypart2 = "0x123123" # whatever

    prog = "/home/eugenek/code/buhacknight/workshops/keygenme-
400/keygenme32.elf"
    args = "{} {} {}".format(username, keypart1, keypart2)
    check_bp = "0x0804A125"

    gdb.execute("file " + prog)
    gdb.execute("b *" + check_bp)
    gdb.execute("r " + args)

    T6 = gdb.parse_and_eval("*(unsigned int*)$esp")
    T7 = gdb.parse_and_eval("*(unsigned int*)($esp + 4)")

    print("T6 = {} T7 = {}".format(T6,T7))
    return (T6, T7)

def crack_keys(T6, T7):
    keypart1 = T6 ^ 0x31333337
    keypart2 = (T7 & 0x000000FF) | (((T7 & 0x00FF0000) >> 16) << 8) | (((T7
& 0xFF000000) >> 24) << 16) | (((T7 & 0x0000FF00) >> 8) << 24)
    return (keypart1, keypart2)

## Talk to the server and get what username it wants
## then send it back the cracked key
s = shoe.Shoe('localhost', 12123)
resp = s.read_until("\r\n") # Welcome msg
for i in range(0,10):
    resp = s.read_until("\n").decode('utf-8') # Username msg
    print(resp)
    username = resp[-23:].rstrip()
    t6, t7 = get_t6t7(username)
    keypart1, keypart2 = crack_keys(t6, t7)

    ans = "{} {}\n".format(str(int(keypart1)), str(int(keypart2)))
    s.write(ans)
    resp = s.read_until("\n").decode('utf-8') # The smiley
    print(resp)

resp = s.read_until("\n") # Let's get our flag!
print(resp)
s.close()
```

Run the script above in `gdb`:

```
$ source pwn.py
give me the password for l5R7Hd06vdzCEgVNBgUS1g
T6 = 719604441 T7 = 1692167297
:-)
give me the password for yhL6Ir0_kkEiIBkqhmJgsQ
T6 = 3377606778 T7 = 1557326192
:-)
... 8 more of these ...
b"here's the flag key{vM_k3yg3n_a1n7_n0_th4ng}\n"
```

Share this post