Added references.
David Manouchehri authored 4 years ago

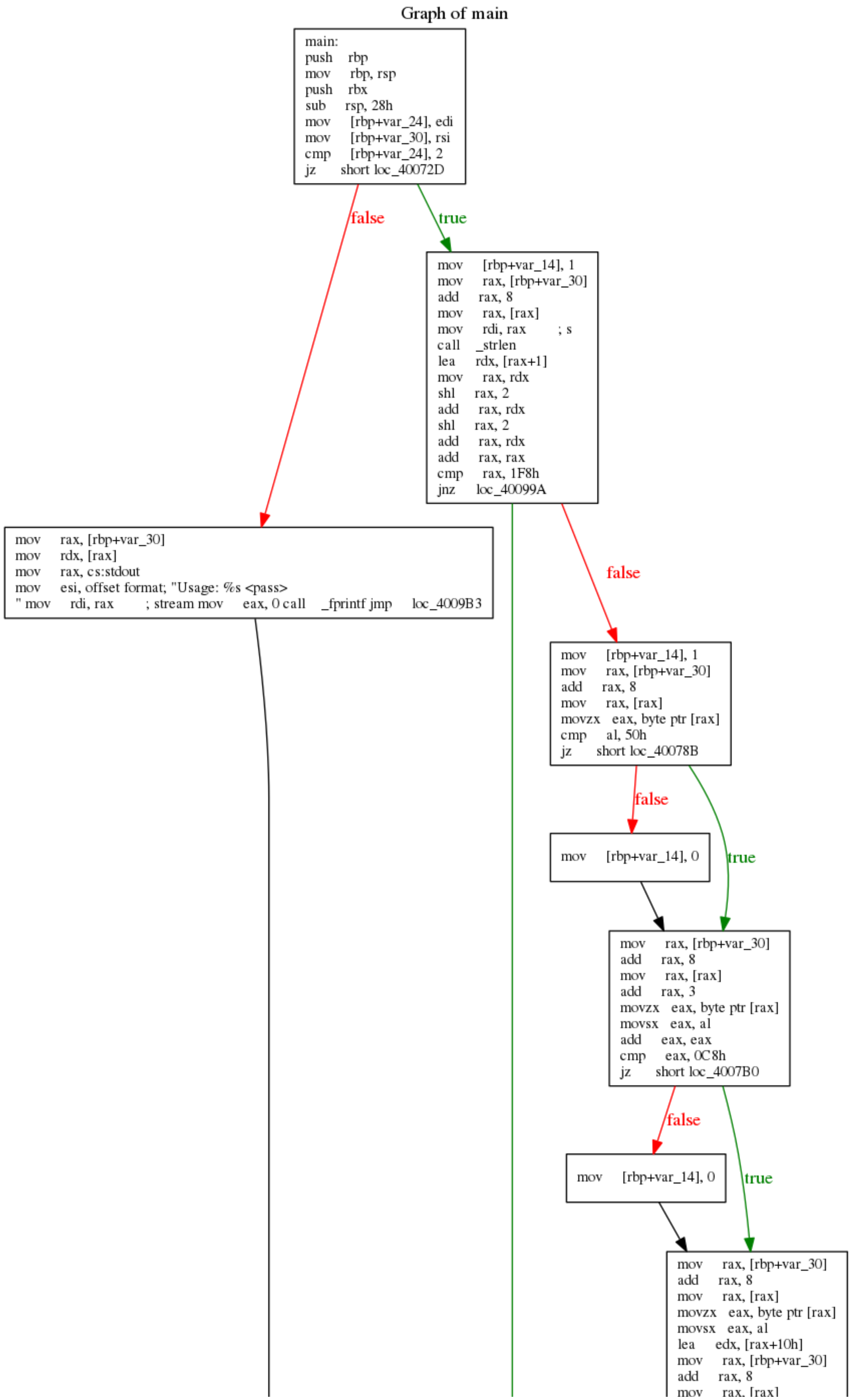
4b645992

stage2.md 6.21 KB

Keygenning with KLEE and Hex-Rays

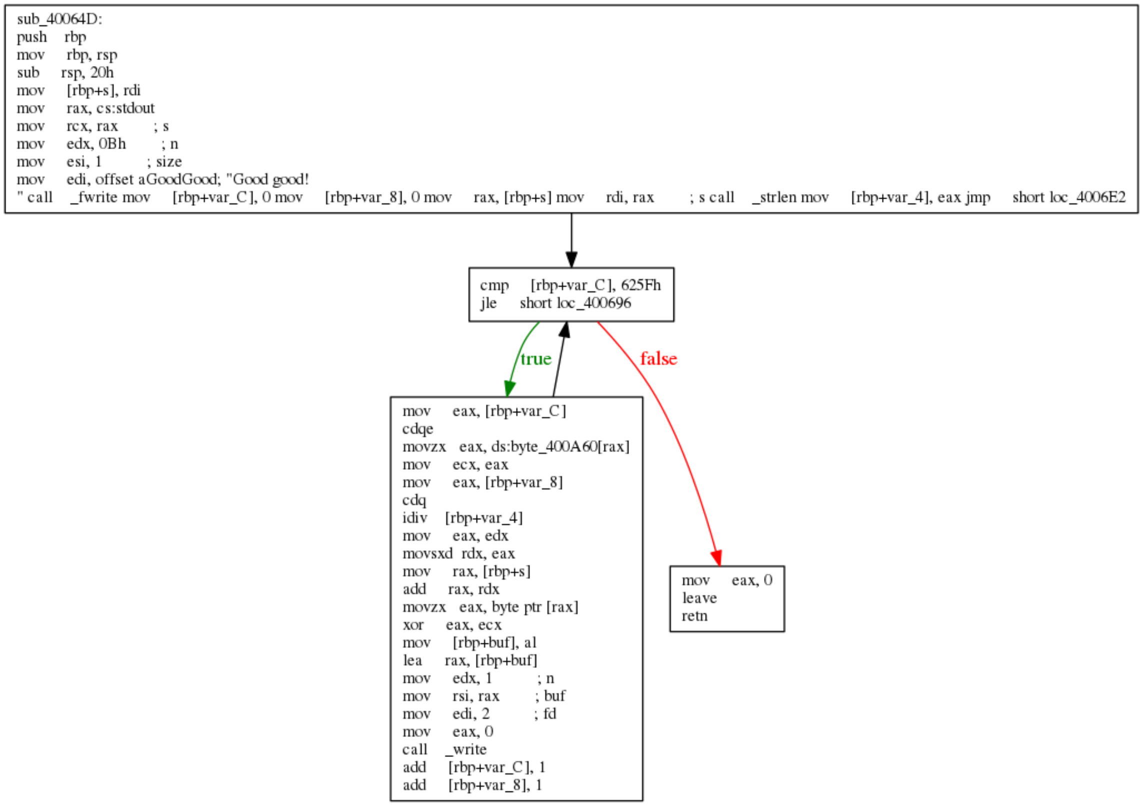
First, we need to identify the code that handles validating passwords. Opening the binary in IDA Pro makes it quite easy to visualize the main function.

Understanding all of the instructions isn't required; it's clear that the left branch occurs (outputting `Usage: %s <pass>`) when no input is give and the middle branch happens when an incorrect password is given (outputting `Try again...`).


Graph of main
main:
push rbp
mov rbp, rsp
push rbx
sub rsp, 28h
mov [rbp+var_24], edi
mov [rbp+var_30], rsi
cmp [rbp+var_24], 2
jz short loc_40072D
false
true
mov [rbp+var_14], 1
mov rax, [rbp+var_30]
add rax, 8
mov rax, [rax]
mov rdi, rax ; s
call _strlen
lea rdx, [rax+1]
mov rax, rdx
shl rax, 2
add rax, rdx
shl rax, 2
add rax, rdx
add rax, rax
cmp rax, 1F8h
jnz loc_40099A
mov rax, [rbp+var_30]
mov rdx, [rax]
mov rax, cs:stdout
mov esi, offset format; "Usage: %s <pass>
" mov rdi, rax ; stream mov eax, 0 call _fprintf jmp loc_4009B3
false
mov [rbp+var_14], 1
mov rax, [rbp+var_30]
add rax, 8
mov rax, [rax]
movzx eax, byte ptr [rax]
cmp al, 50h
jz short loc_40078B
false
true
mov [rbp+var_14], 0
mov rax, [rbp+var_30]
add rax, 8
mov rax, [rax]
add rax, 3
movzx eax, byte ptr [rax]
movsx eax, al
add eax, eax
cmp eax, 0C8h
jz short loc_4007B0
false
true
mov [rbp+var_14], 0
mov rax, [rbp+var_30]
add rax, 8
mov rax, [rax]
movzx eax, byte ptr [rax]
movsx eax, al
lea edx, [rax+10h]
mov rax, [rbp+var_30]
add rax, 8
mov rax, [rax]

On the bottom right, there's a call to sub_40064D that *should* be for the correct. That being said, I've seen some CTF challenges where there's actually no 'correct' password, so let's look at the function to make sure this isn't a trick.

Graph of sub_40064D



Lucky for us, there's no tricks. Using Hex-Rays Decompiler on main results in extremely readable C.

```
int __fastcall main(int a1, char **a2, char **a3)
{
  int result; // eax@2
  __int64 v4; // rbx@10
  signed int v5; // [sp+1Ch] [bp-14h]@4

  if ( a1 == 2 )
  {
    if ( 42 * (strlen(a2[1]) + 1) != 504 )
      goto LABEL_31;
    v5 = 1;
    if ( *a2[1] != 80 )
      v5 = 0;
    if ( 2 * a2[1][3] != 200 )
      v5 = 0;
    if ( *a2[1] + 16 != a2[1][6] - 16 )
      v5 = 0;
    v4 = a2[1][5];
    if ( v4 != 9 * strlen(a2[1]) - 4 )
      v5 = 0;
    if ( a2[1][1] != a2[1][7] )
      v5 = 0;
    if ( a2[1][1] != a2[1][10] )
      v5 = 0;
    if ( a2[1][1] - 17 != *a2[1] )
      v5 = 0;
```

```
    if ( a2[1][3] != a2[1][9] )
      v5 = 0;
    if ( a2[1][4] != 105 )
      v5 = 0;
    if ( a2[1][2] - a2[1][1] != 13 )
      v5 = 0;
    if ( a2[1][8] - a2[1][7] != 13 )
      v5 = 0;
    if ( v5 )
      result = sub_40064D(a2[1]);
    else
LABEL_31:
      result = fprintf(stdout, "Try again...\n", a2);
  }
  else
  {
    result = fprintf(stdout, "Usage: %s <pass>\n", *a2, a2);
  }
  return result;
}
```

At this point, you can solve the equations by hand in a minute or two by looking up all the ASCII values. However, I decided that wouldn't be a very fun learning experience, and decide to automate the process.

First, main.c needs to be adjusted slightly to be able to be recompiled. Only 2 lines were added and 7 changed.

```
#include "defs.h" // Take from IDA's plugins/
#include <string.h>
#include <stdio.h>

int main(int a1, char **a2, char **a3)
{
  __int64 v4; // rbx@10
  signed int v5; // [sp+1Ch] [bp-14h]@4

  if ( a1 == 2 )
  {
    if ( 42 * (strlen(a2[1]) + 1) != 504 )
      goto LABEL_31;
    v5 = 1;
    if ( *a2[1] != 80 )
      v5 = 0;
    if ( 2 * a2[1][3] != 200 )
      v5 = 0;
    if ( *a2[1] + 16 != a2[1][6] - 16 )
      v5 = 0;
    v4 = a2[1][5];
    if ( v4 != 9 * strlen(a2[1]) - 4 )
      v5 = 0;
    if ( a2[1][1] != a2[1][7] )
      v5 = 0;
    if ( a2[1][1] != a2[1][10] )
      v5 = 0;
    if ( a2[1][1] - 17 != *a2[1] )
      v5 = 0;
    if ( a2[1][3] != a2[1][9] )
      v5 = 0;
    if ( a2[1][4] != 105 )
      v5 = 0;
    if ( a2[1][2] - a2[1][1] != 13 )
      v5 = 0;
    if ( a2[1][8] - a2[1][7] != 13 )
```

```
      v5 = 0;
    if ( v5 )
      printf("Good good!\n");
    else
LABEL_31:
      printf("Try again...\n");
  }
  else
  {
    printf("Usage: %s <pass>\n", *a2);
  }
}
```

To solve the program, we can use KLEE (which is a symbolic virtual machine). I opted to use the [Docker image of KLEE](#) to avoid building it.

```
sudo docker pull klee/klee
sudo docker run -v /mnt/hgfs/stage2:/home/klee/stage2 --rm -ti --ulimit='stack=-1:-1' klee/klee # Adjust -v
```

To flag to KLEE when we've reached the state we want, `klee_assert(0)` has to be added (which will flag it down as an error when it occurs).

```
#include "defs.h" // Take from IDA's plugins/
#include <string.h>
#include <stdio.h>
#include <assert.h>
#include <klee/klee.h>

int main(int a1, char **a2, char **a3)
{
  __int64 v4; // rbx@10
  signed int v5; // [sp+1Ch] [bp-14h]@4

  if ( a1 == 2 )
  {
    if ( 42 * (strlen(a2[1]) + 1) != 504 )
      goto LABEL_31;
    v5 = 1;
    if ( *a2[1] != 80 )
      v5 = 0;
    if ( 2 * a2[1][3] != 200 )
      v5 = 0;
    if ( *a2[1] + 16 != a2[1][6] - 16 )
      v5 = 0;
    v4 = a2[1][5];
    if ( v4 != 9 * strlen(a2[1]) - 4 )
      v5 = 0;
    if ( a2[1][1] != a2[1][7] )
      v5 = 0;
    if ( a2[1][1] != a2[1][10] )
      v5 = 0;
    if ( a2[1][1] - 17 != *a2[1] )
      v5 = 0;
    if ( a2[1][3] != a2[1][9] )
      v5 = 0;
    if ( a2[1][4] != 105 )
      v5 = 0;
    if ( a2[1][2] - a2[1][1] != 13 )
      v5 = 0;
    if ( a2[1][8] - a2[1][7] != 13 )
      v5 = 0;
    if ( v5 ) {
```

```
      printf("Good good!\n");
      klee_assert(0);
    }
    else
LABEL_31:
      printf("Try again...\n");
  }
  else
  {
    printf("Usage: %s <pass>\n", *a2);
  }
}
```

Onto compiling!

`clang -I ~/klee_src/include/ -emit-llvm -g -o main.ll -c main.`

Now, here's the magic of KLEE. Running `klee --optimize --libc=uclibc --posix-runtime main.ll --sym-arg 100` will look for all possible solutions that are under 100 chars. After a mere few seconds (on an ancient mobile i3 CPU), the entire process is done. In this binary there's one `*.err` file since there's only one solution.

```
klee@4a860a030d10:~/stage2$ ls klee-last/*err
klee-last/test000025.assert.err

klee@4a860a030d10:~/stage2$ ktest-tool klee-last/test000025.ktest
ktest file : 'klee-last/test000025.ktest'
args       : ['main.ll', '--sym-arg', '100']
num objects: 2
object    0: name: b'arg0'
object    0: size: 101
object    0: data: b'Pandi_panda\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xf
object    1: name: b'model_version'
object    1: size: 4
object    1: data: b'\x01\x00\x00\x00'
```

We can see that the input Pandi_panda (followed by a null byte) is the correct password.

References

- https://doar-e.github.io/blog/2015/08/18/keygenning-with-klee/
- https://klee.github.io/tutorials/testing-function/
- https://klee.github.io/tutorials/testing-coreutils/
- https://www.cs.umd.edu/~mwh/se-tutorial/
- https://www.cs.purdue.edu/homes/kim1051/cs490/proj3/description.html