A solution to “easycrackme” by “bxplode”

A. S. “Aleksey” Ahmann*

November 22, 2024

Contents

1 Introduction

On the `crackmes.one` website, user “bxplode” published an easy, high quality,[1] “crack me” toy problem [1]. End-users are encouraged to work out a solution and submit their findings without the use of binary patching.[2] Here, the toy problem comes in the form of a binary executable where its respective solution comes in the form of a “key” that will cause the application to output a message affirming that a correct key has been worked out.

*Relevant contacts and identifiers:
Email: hackermaneia@riseup.net
`crackmes.one` Account: https://crackmes.one/user/RelationalAlgebra
GitHub Portfolio: https://github.com/Alekseyyy

[1]This “crack me” has a difficulty score of 1.4/6, and a quality score of 5/6.

[2]See the *crackmes.one* FAQ.

Before trying to find a solution, I should begin by setting up the problem. I proceeded by downloading the crack me ZIP archive and extracting the files.[3] The relevant file in the archive is `bxtumations_crackme.exe`. I ran the software executable, which gave me the prompt depicted in figure 1:

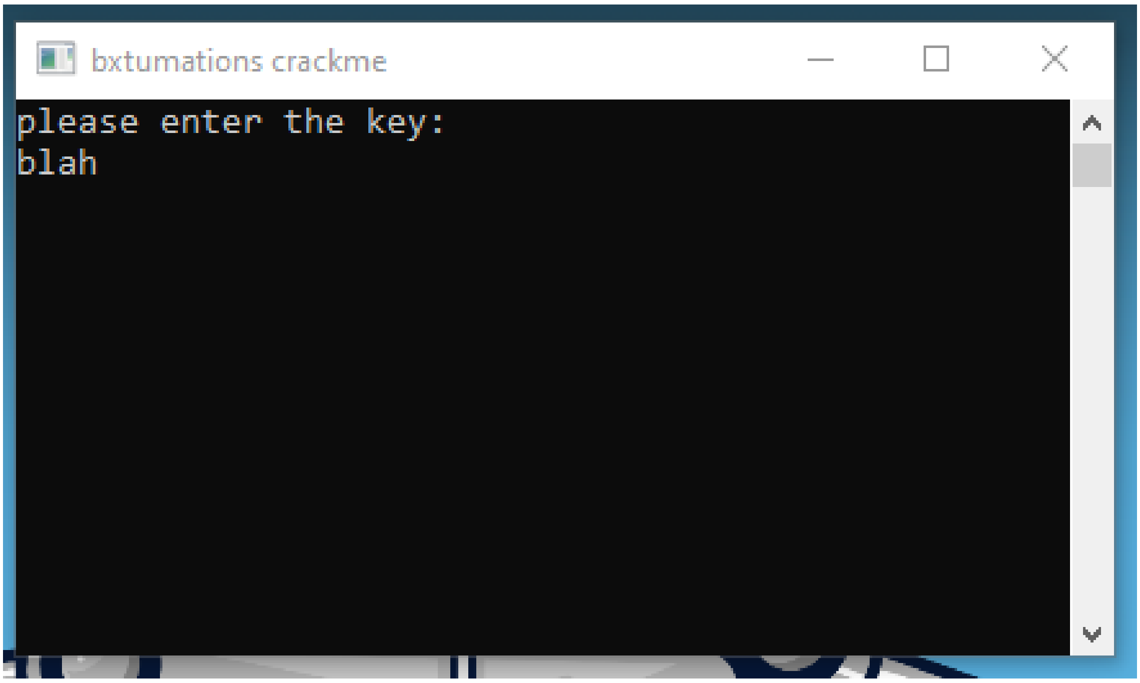

Figure 1: Prompt given when running `bxtumations_crackme.exe`

I typed in "`blah`" followed by the `enter`-key, and was presented with a message informing me that my solution was incorrect (figure 2). Hereforth, the problem defined is to work out a string that, when inputted into the application, will result in a message that is not "`wrong key!!`" and "`try again :))`".

2 Solution: Static and Dynamic Analysis

I will go about working out the proper key with the methods of static and dynamic analysis.[4] This section will outline how I went about deriving the solution.

2.1 Initial Analysis of the Software System

The first step that I took is to load `bxtumations_crackme.exe` into a number of software reverse engineering tools, such as *Ghidra* [2], *IDA Pro*[5] [3], *Detect It Easy* [4], and *x64dbg* [5]. `crackmes.one` listed this binary as being written in C/C++ and written for Microsoft Windows. The *Detect It Easy* results (figure 3) confirms this.

[3] The password for the ZIP archive is "crackmes.one"
[4] In a future writeup, I intend to discuss an alternative solution.
[5] Free Edition

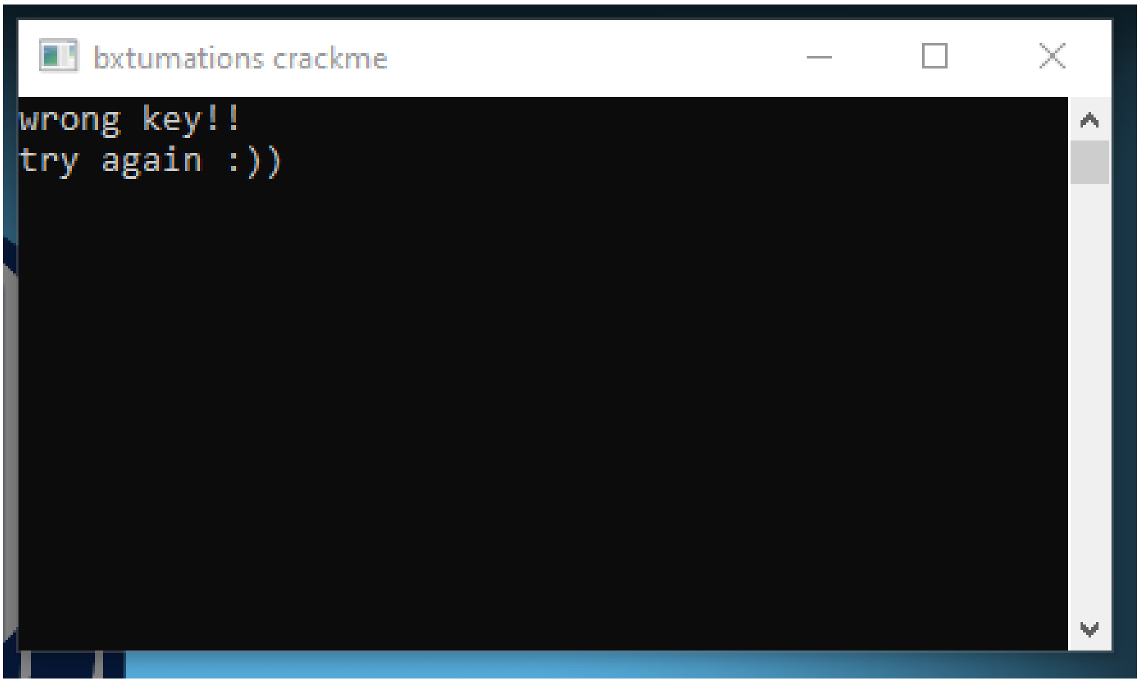

Figure 2: Error message stating that `blah` is not the correct key

When loading `bxtumations_crackme.exe` into *Ghidra*, I made sure that the format was set to "Portable Executable (PE)" and used all of Ghidra's "analysis" options on it.[6] A cursory look at the number of calls that the executable makes tells me that trying to trace its execution flow by manually graphing it would take too long. So, I instead decided to try to identify the part of the executable where the key is worked out and stored in memory.

Using Ghidra's export features, I had its decompiler save a high-level C/C++ representation of the binary executable onto disk.[7] I then used a standard text editor to search for the string `please enter the key` — where I discovered that the key is worked out in the `FUN_00401789` function.

I decided to give a cursory look at `FUN_00401789`, and the following is a snippet of the relevant source code:

```
undefined8 FUN_00401789(void){

  undefined8 uVar1;

  [... snip ...]

  while( true ) {
    FUN_004a0e90(&DAT_004a6860,local_c8);
    ppvVar3 = local_108;
    pplVar2 = local_c8;
```

[6]Including the beta/experimental analysers.

[7]This is done by going to `File > Export Program`, setting the "format" to C/C++, and specifiying a location to save the decompile dump.

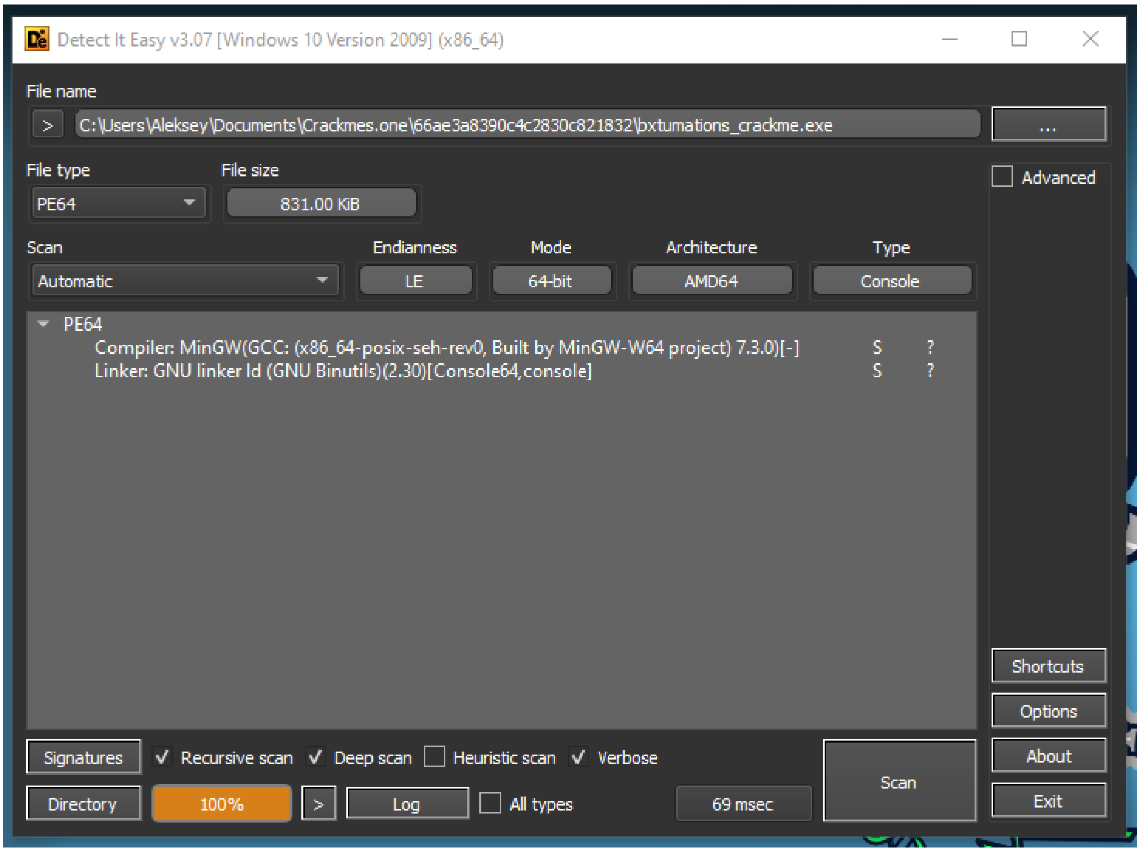

Figure 3: *Detect-It-Easy* Output

```
        uVar1 = FUN_0049fbe0(pplVar2,ppvVar3);
        if ((char)uVar1 != '\0') break;
        FUN_00401560();
        FUN_00460b70();

    [... snip ...]

    }
    DAT_004d1030 = 1;
    FUN_0040157b(pplVar2,ppvVar3,ppiVar4);
    FUN_00491060(local_108);
    FUN_00491060(local_e8);
    FUN_00491060(local_c8);
    return 0;
}
```

I removed a lot of what I see to be irrelevant information, but the reader may consult the "supplementary materials" and the Appendix[8] if they want the full source code. This relevant bits of the function works as follows:

[8]Under the section title "C/C++ Decompile Dumps"

1. First, a variable called `uVar1` of type “`undefined8`” is declared.[9] Other variables are declared, but I am not interested in them.

2. Next, other functions are called, and then the program goes to an infinite `while`-loop.

3. In this `while`-loop, the `uVar1` is set to the results of `FUN_0049fbe0(pplVar2,ppvVar3)`, and then it is used in a conditional.

4. Regarding the conditional, the `uVar1` is casted to the `char` type — `(char)uVar1` — and then compared to the character `\0`.

   (a) If `uVar1` is equal to `\0`, then the program will `break` out of the `while`-loop.

   (b) Otherwise, the program will display a “wrong key” error message, and not break out of the while-loop.

5. Assuming that the program breaks out of the `while`-loop, a message will be printed onto the screen congratulating the end-user for working out the correct key.

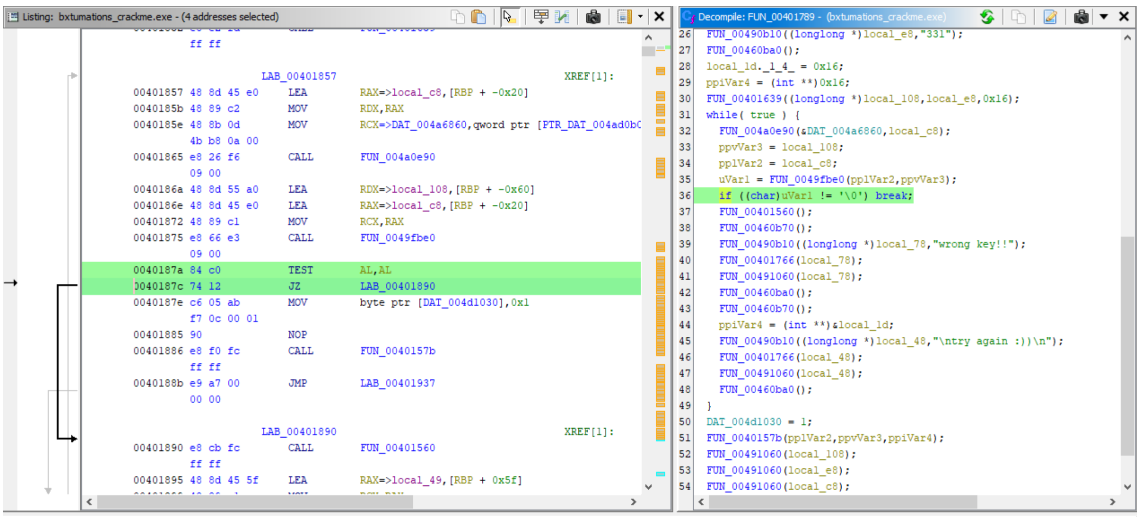

Figure 4: A comparison of `FUN_00401789`’s C/C++ representation to its disassembly

I focused on the disassembly of the conditional. On *Ghidra*, I compared the C/C++ decompile dump of the binary to its respective disassembly — which is depicted by figure 4 depicts this. In particular, the `if ((char)uVar1 != ’\0’) break;` corresponds to the following assembler instructions:

```
1. TEST AL,AL
2. JZ LAB_00401890
```

[9] I think that this might be an `unsigned int`, though I was told that it could be some 8-bit type.

Line 1 has an address value of `0040187a` and line 2 has an address value of `0040187c`. This will become relevant as I move on to dynamic analysis with the *x64dbg* debugger.

2.2 Breakpoints and Debugger Setup

The software binary is too complicated for me to understand with just the methods of manually charting the program's execution flow with a directed graph. The aforementioned method would take too long, and if I assume that "time is of the essence," then I should find a quicker way to work out a solution. This "quicker way" involves the *x64dbg* debugger: after loading the `bxtumations_crackme.exe` into it, I was presented with four panes showing states and information regarding the software binary pre-execution, and during execution — as depicted by figure 5.

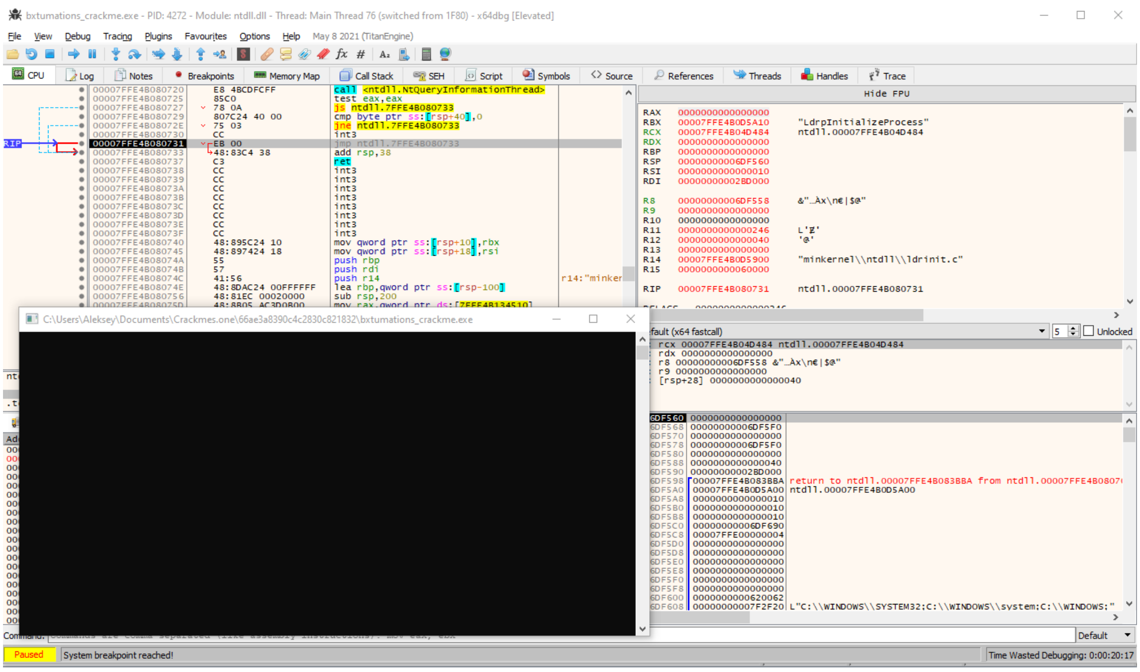

Figure 5: *x64dbg* view after loading `bxtumations_crackme.exe`

I proceeded by identifying the assembly instructions with the address values `0040187a` and `0040187c`, and proceeding to set breakpoints onto them — which is depicted by figure 6. I expect that, when I run the software under the debugger, it will run the necessary calculations and "self-decode" the key, and then load the key into memory, CPU registers, or other kinds of memory. I was right, and will discuss my findings in the next subsection.

2.3 Discovering the Key

After I have configured the breakpoints on the *x64dbg* debugger, I ran the executable, and it paused execution on the breakpoints. Figure 7 depicts what each of *x64dbg*'s four panes looked like after running until it reached the breakpoints.

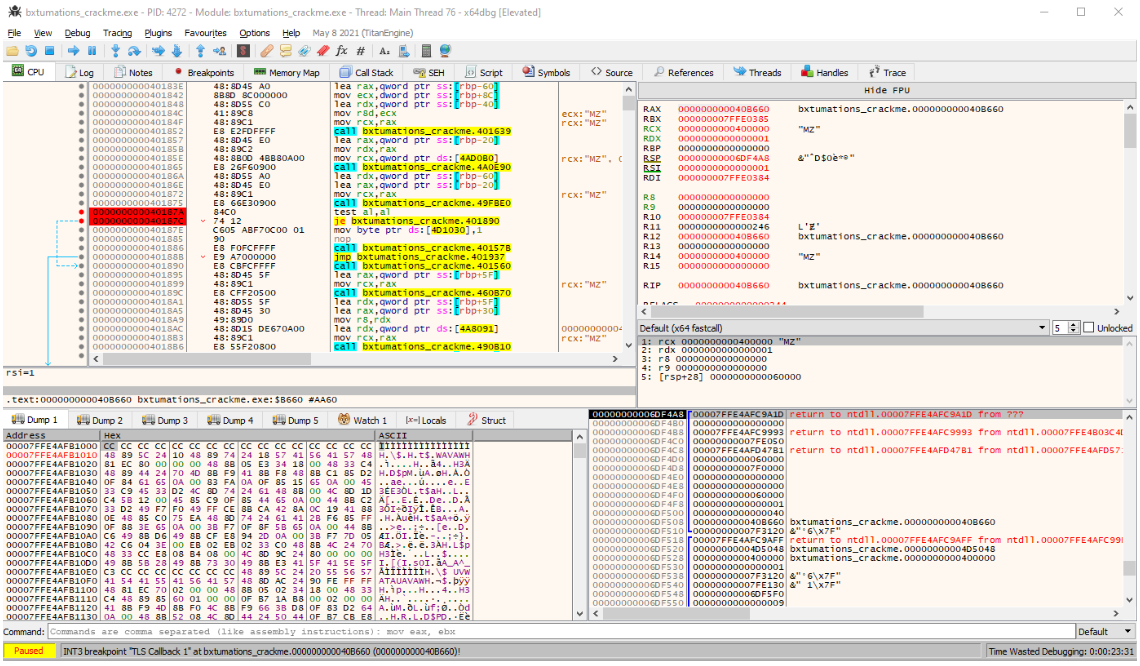

Figure 6: Setting breakpoints on 0040187a and 0040187c

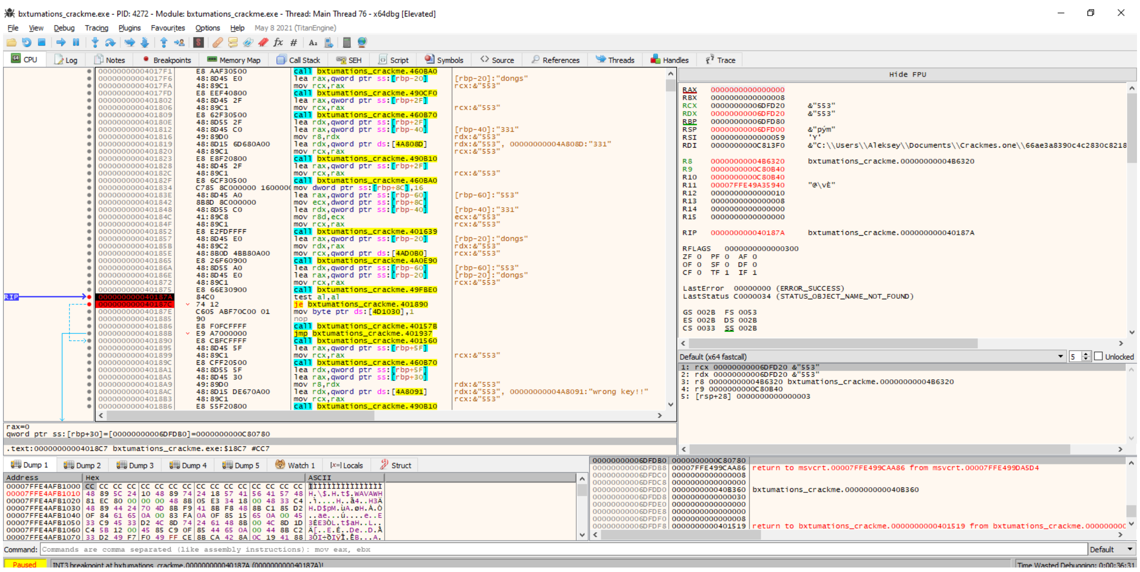

Figure 7: Execution state up until the set breakpoints (on 0040187a and 0040187c)

The following is some of what was reported on the top-right pane showing CPU register values:

```
RAX    0000000000000000
RBX    0000000000000008
RCX    00000000006DFD20    &"553"
RDX    00000000006DFD20    &"553"
RBP    00000000006DFD80
RSP    00000000006DFD00    &"pym"
```

```
[... snip ...]
```

Furthermore, the following was reported on the top-left pane showing the software binary's disassembly.

```
[... snip ...]

00401857    lea rax, qword ptr ss:[rbp-20] ; [rbp-20]:"dongs"
0040185B    mov rdx, rax ; rdx:&"553"
00401853    mov rcx, qword ptr ds:[4AD0B0] ; rcx:&"553"
00401865    call bxtumations_crackme.4A0E90
0040186A    lea rdx, qword ptr ss:[rbp-60] ; [rbp-50]:"553"
0040186E    lea rax, qword ptr ss:[rbp-20] ; [rbp-20]:"dongs"
00401872    mov rcx, rax ; rcx:"553"
00401875    call bxtumations_crackme.49FBE0
0040187A    test al, al
0040187C    je bxtumations_crackme.401890

[... snip ...]
```

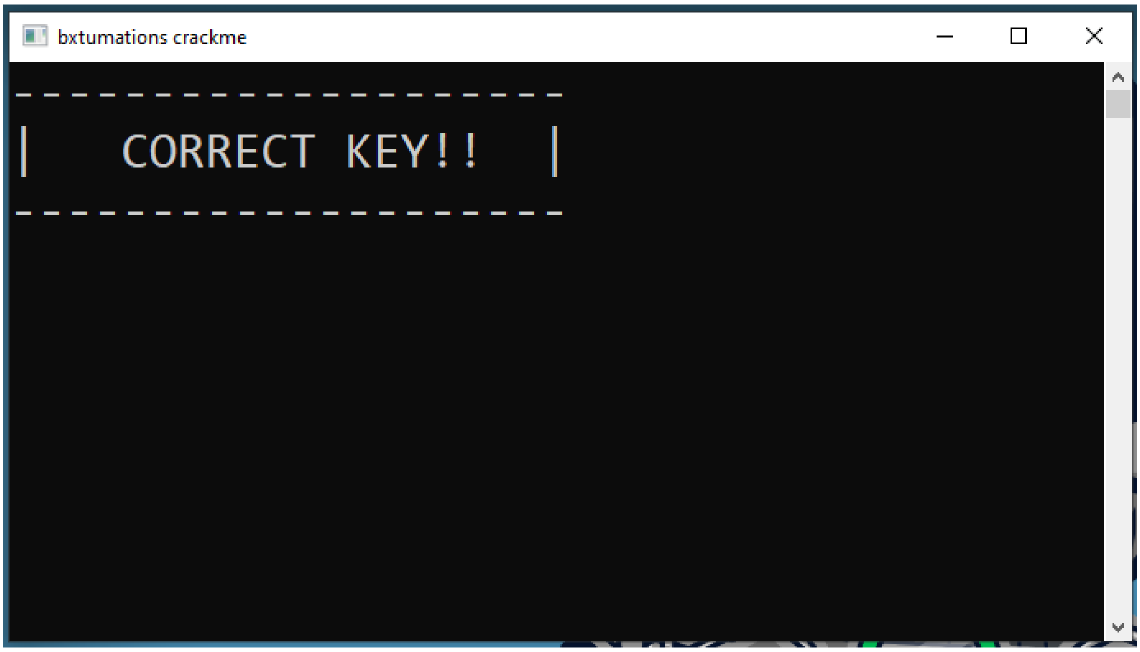

Figure 8: Message when `553` is entered as the key.

From an intuitive look at both the registers, stack values, and disassembler, I guessed that "553" is the correct key. I ran `bxtumations_crackme.exe` without a debugger, typed in "553", and was presented with a message stating that I have supplied the "correct key" — as depicted by figure 8.[10]

[10] Unlike with the first two figures, I made the text bigger by changing the font settings in the command prompt.

While I initially worked out the key with a combination of intuition, experimentation, and luck, I do think that I should try to analyse *why* this is the correct key. Looking back at the CPU registers, I noticed that the number `553`, represented as a string, has its address referenced into the `RCX` and `RDX` registers. I also noticed the following instructions[11] of the CPU assembler dump shown earlier:

```
[... snip ...]

05. 0040186A    lea rdx, qword ptr ss:[rbp-60] ; [rbp-50]:"553"
06. 0040186E    lea rax, qword ptr ss:[rbp-20] ; [rbp-20]:"dongs"
07. 00401872    mov rcx, rax ; rcx:"553"
08. 00401875    call bxtumations_crackme.49FBE0
09. 0040187A    test al, al
10. 0040187C    je bxtumations_crackme.401890
```

The fifth line[12] references "`553`"'s address in the `RDX` register, and then the sixth line[13] references the `dongs` string's address into the `RAX` register. This `RAX` address is then copied in the `RCX` register,[14] and then a function is called.[15] I am not quite sure what that function does, but the `test` instruction is executed in the ninth line,[16] and the *jump if equal* instruction is executed on the tenth line.[17] This tells me that the string `553` is being compared to the input, and serves as a useful hint for guessing how it could affect the trajectory of the software binary's execution path.

3 End Matter

By the procedure outlined in the previous section, I can confidently say that "`553`" is the correct key. The following are "takeaways" that I have learnt from doing this short project:

- Given that certain assumptions are made,[18] static analysis is useful for mapping out how a software binary's system looks like.
- Regarding static analysis, a "easy-to-read" decompiler output can be compared to a more concrete assembler output to find clues when setting debugger breakpoints.

[11] Lines 05-10, or addresses `0040186A` to `0040187C`.

[12] Ln. 05

[13] Ln. 06

[14] Ln. 07

[15] Ln. 08

[16] Ln. 09

[17] Ln. 10

[18] For example, in a criticism of a malware analysis report [6], Robert Graham states that reverse engineering tools make assumptions about a binary, and the reverse engineer or systems analyst must use intuition and good judgement for determining if assumptions are met.

- Software can be very complex,[19] even when it is the "easy" `crackmes.one` puzzle that is the subject of this writeup. Dynamic analysis done by a debugger can help reverse engineers "cut through the complexity" to solve a problem. In this case, I used a debugger to run the software binary and observe CPU registers and the software's disassembly, which allowed me to work out the correct key.

- While this paper is not technically "original research," it does share at most a few things in common with it: mainly that they both do not have a "fixed answer.[20]" Academic research, does not always give "clean" results, and sometimes guesswork and intuition is needed to interpret results and observation. But nonetheless, a well defined criteria for progress is needed for research to be successful.

3.1 Supplementary Materials

The project files can be accessed from the following GitHub repository:

https://github.com/Alekseyyy/SNHU/tree/main/sundries/wargames/crackmes.one/writeups/66ae3a8390c4c2830c821832

3.2 About the Author

At the time of this writing, I am a junior computer science undergraduate student, with a minor in mathematics and a concentration in data analysis. I currently do bug-bounty and responsible vulnerability disclosure. I also enjoy learning more about low-level aspects of computer hardware and assembler languages by solving toy-problems and through resources outside of school.[21]

3.3 Acknowledgements

I would like to take the time to acknowledge the helpful feedback from the following on earlier drafts of this paper:

- *Anonymous collaborator 1* for encouraging me to expand more on bulletpoints in the conclusion.

- *Anonymous collaborator 2* for informing me that an "undefined" data type might be an 8-bit data type (as opposed to specifically an `unsigned int`), for helping me better articulate how stack values get referenced in memory, and for helping me better articulate my points in general.

[19]In a paper [7] by security researcher Greg Hoglund, he discusses how software that is made up of just simple rules can nonetheless produce complex behaviour, and how such complexity can lead to software bugs.

[20]I was informed by *Anonymous collaborators 2 and 3* that this writeup resembles more of a homework problem than a original research.

[21]Like by solving `crackmes.one` puzzles ;-)

- *Anonymous collaborator 3* for convincing me that the analogy between `crackmes.one` puzzles and origial research has severe limitations.

I take full responsibility for any erratas in this writeup.

A C/C++ Decompile Dumps

I included the following decompile dumps in this writeup and in the supplementary materials:

- `entry.c`: this is the entry point of the application, which executes the code blocks represented by C functions: `FUN_0040b450` and `FUN_00401180`.
- `FUN_0040b450.c`: this is one of the functions called by the entry point (I don't know exactly what it does).
- `FUN_00401180.c`: this is another one of the functions called by the entry point (I don't know exactly what it does).
- `FUN_00401789.c`: this is the function that I looked at when trying to work out what addresses to set breakpoints when debugging a the software binary. I documented a simplified version of it in the section where I looked at the decompile dump of the software binary.

A.1 `entry.c`

```
void entry(undefined8 param_1,
  undefined8 param_2,undefined8 param_3)

{
  DAT_004d1610 = 0;
  FUN_0040b450();
  FUN_00401180(param_1,param_2,param_3);
  return;
}
```

A.2 `FUN_0040b450.c`

```
void FUN_0040b450(void)

{
  _FILETIME _Var1;
  DWORD DVar2;
  DWORD DVar3;
  DWORD DVar4;
  _FILETIME local_38;
```

```
  LARGE_INTEGER local_30;

  local_38.dwLowDateTime = 0;
  local_38.dwHighDateTime = 0;
  if (DAT_004a7200 != 0x2b992ddfa232) {
    DAT_004a7210 = ~DAT_004a7200;
    return;
  }
  GetSystemTimeAsFileTime(&local_38);
  _Var1 = local_38;
  DVar2 = GetCurrentProcessId();
  DVar3 = GetCurrentThreadId();
  DVar4 = GetTickCount();
  QueryPerformanceCounter(&local_30);
  DAT_004a7200 = ((ulonglong)DVar4 ^
    (ulonglong)DVar3 ^ (ulonglong)DVar2 ^ (ulonglong)_Var1
    ^ local_30.QuadPart) & 0xffffffffffff;
  if (DAT_004a7200 == 0x2b992ddfa232) {
    DAT_004a7210 = 0xffffd466d2205dcc;
    DAT_004a7200 = 0x2b992ddfa233;
  }
  else {
    DAT_004a7210 = ~DAT_004a7200;
  }
  return;
}
```

A.3 FUN_00401180.c

```
/* WARNING: Globals starting with '_'
  overlap smaller symbols at the same address */

ulonglong FUN_00401180(undefined8 param_1,
  undefined8 param_2,undefined8 param_3)

{
  int iVar1;
  void **ppvVar2;
  char cVar3;
  ulonglong uVar4;
  ulonglong uVar5;
  undefined8 *puVar6;
  int iVar7;
  char **ppcVar8;
  char *pcVar9;
```

```
  undefined8 *puVar10;
  size_t sVar11;
  void *_Dst;
  undefined8 *puVar12;
  ulonglong uVar13;
  longlong lVar14;
  undefined8 uVar15;
  undefined8 uVar16;
  LPSTARTUPINFOA p_Var17;
  undefined8 uVar18;
  longlong unaff_GS_OFFSET;
  bool bVar19;
  undefined local_a8 [64];
  ushort local_68;

  p_Var17 = (LPSTARTUPINFOA)local_a8;
  for (lVar14 = 0xd; lVar14 != 0; lVar14 = lVar14 + -1) {
    *(undefined8 *)p_Var17 = 0;
    p_Var17 = (LPSTARTUPINFOA)&p_Var17->lpReserved;
  }
  uVar13 = (ulonglong)DAT_004d1610;
  if (DAT_004d1610 != 0) {
    GetStartupInfoA((LPSTARTUPINFOA)local_a8);
  }
  uVar4 = *(ulonglong *)(*(longlong *)(unaff_GS_OFFSET
    + 0x30) + 8);
  while( true ) {
    LOCK();
    bVar19 = DAT_004d2420 == 0;
    DAT_004d2420 = DAT_004d2420 ^ (ulonglong)bVar19 *
      (DAT_004d2420 ^ uVar4);
    uVar5 = !bVar19 * DAT_004d2420;
    UNLOCK();
    if (uVar5 == 0) break;
    if (uVar4 == uVar5) {
      bVar19 = true;
      goto joined_r0x004011ff;
    }
    Sleep(1000);
  }
  bVar19 = false;
joined_r0x004011ff:
  if (DAT_004d2428 == 1) {
    _amsg_exit(0x1f);
  }
  else if (DAT_004d2428 == 0) {
```

```
    DAT_004d2428 = 1;
    _initterm();
  }
  else {
    DAT_004d1004 = 1;
  }
  if (DAT_004d2428 == 1) {
    _initterm();
    DAT_004d2428 = 2;
  }
  if (!bVar19) {
    LOCK();
    DAT_004d2420 = 0;
    UNLOCK();
  }
  uVar18 = 0;
  uVar16 = 2;
  uVar15 = 0;
  tls_callback_0(0,2);
  FUN_0040ba50(uVar15,uVar16,uVar18,uVar13);
  DAT_004d1640 = SetUnhandledExceptionFilter(
    (LPTOP_LEVEL_EXCEPTION_FILTER)&LAB_0040bfa0);
  FUN_0040beb0();
  FUN_004175f0(&LAB_00401000);
  FUN_0040b850();
  _DAT_004d2410 = &IMAGE_DOS_HEADER_00400000;
  ppcVar8 = (char **)FUN_004176d0();
  iVar7 = DAT_004d1020;
  bVar19 = false;
  pcVar9 = *ppcVar8;
  if (pcVar9 != (char *)0x0) {
    do {
      cVar3 = *pcVar9;
      if (cVar3 < '!') {
        if ((cVar3 == '\0') || (!bVar19)) goto LAB_004012d0;
        bVar19 = true;
      }
      else if (cVar3 == '\"') {
        bVar19 = (bool)(bVar19 ^ 1);
      }
      pcVar9 = pcVar9 + 1;
    } while( true );
  }
  goto LAB_004012f7;
LAB_004012d0:
  _DAT_004d2418 = pcVar9;
```

```
  if (cVar3 != '\0') {
    do {
      pcVar9 = pcVar9 + 1;
      _DAT_004d2418 = pcVar9;
      if (*pcVar9 == '\0') break;
    } while (*pcVar9 < '!');
  }
LAB_004012f7:
  if ((DAT_004d1610 != 0) &&
    (_DAT_004a4000 = 10, (local_a8[60] & 1) != 0)) {
    _DAT_004a4000 = (uint)local_68;
  }
  iVar1 = DAT_004d1020 + 1;
  puVar10 = (undefined8 *)malloc((longlong)iVar1 * 8);
  puVar6 = DAT_004d1018;
  puVar12 = puVar10;
  if (0 < iVar7) {
    lVar14 = 0;
    do {
      sVar11 = strlen(*(char **)((longlong)puVar6 + lVar14));
      _Dst = malloc(sVar11 + 1);
      *(void **)((longlong)puVar10 + lVar14) = _Dst;
      ppvVar2 = (void **)((longlong)puVar6 + lVar14);
      lVar14 = lVar14 + 8;
      memcpy(_Dst,*ppvVar2,sVar11 + 1);
    } while ((ulonglong)(iVar7 - 1) * 8 + 8 != lVar14);
    puVar12 = puVar10 + (longlong)iVar1 + -1;
  }
  *puVar12 = 0;
  DAT_004d1018 = puVar10;
  FUN_0040b410();
  *(undefined8 *)__initenv_exref = DAT_004d1010;
  uVar13 = FUN_00401789();
  DAT_004d100c = (uint)uVar13;
  if (DAT_004d1008 != 0) {
    if (DAT_004d1004 == 0) {
      _cexit();
      uVar13 = (ulonglong)DAT_004d100c;
    }
    return uVar13;
  }
  /* WARNING: Subroutine does not return */
  exit(DAT_004d100c);
}
```

A.4 FUN_00401789.c

```
undefined8 FUN_00401789(void)

{
  undefined8 uVar1;
  longlong **pplVar2;
  void **ppvVar3;
  int **ppiVar4;
  void *local_108 [4];
  void *local_e8 [4];
  longlong *local_c8 [4];
  void *local_a8 [6];
  void *local_78 [6];
  void *local_48 [5];
  undefined8 local_1d;

  FUN_0040b410();
  SetConsoleTitleA("bxtumations crackme");
  FUN_00460b70();
  FUN_00490b10((longlong *)local_a8,"please enter the key:\n");
  FUN_00401766(local_a8);
  FUN_00491060(local_a8);
  FUN_00460ba0();
  FUN_00490cf0((longlong *)local_c8);
  FUN_00460b70();
  FUN_00490b10((longlong *)local_e8,"331");
  FUN_00460ba0();
  local_1d._1_4_ = 0x16;
  ppiVar4 = (int **)0x16;
  FUN_00401639((longlong *)local_108,local_e8,0x16);
  while( true ) {
    FUN_004a0e90(&DAT_004a6860,local_c8);
    ppvVar3 = local_108;
    pplVar2 = local_c8;
    uVar1 = FUN_0049fbe0(pplVar2,ppvVar3);
    if ((char)uVar1 != '\0') break;
    FUN_00401560();
    FUN_00460b70();
    FUN_00490b10((longlong *)local_78,"wrong key!!");
    FUN_00401766(local_78);
    FUN_00491060(local_78);
    FUN_00460ba0();
    FUN_00460b70();
    ppiVar4 = (int **)&local_1d;
    FUN_00490b10((longlong *)local_48,"\ntry again :))\n");
    FUN_00401766(local_48);
```

```
    FUN_00491060(local_48);
    FUN_00460ba0();
  }
  DAT_004d1030 = 1;
  FUN_0040157b(pplVar2,ppvVar3,ppiVar4);
  FUN_00491060(local_108);
  FUN_00491060(local_e8);
  FUN_00491060(local_c8);
  return 0;
}
```

References

[1] "bxplode," "easycrackme," crackmes.one https://crackmes.one/crackme/66ae3a8390c4c2830c821832 (accessed Nov. 16, 2024)

[2] Ghidra, https://ghidra-sre.org/ (accessed Nov. 19, 2024).

[3] "IDA Pro (Free)," ida-free, https://hex-rays.com/ida-free (accessed Nov. 19, 2024).

[4] "Horsicq," "Horsicq/detect-it-easy: Program for determining types of files for windows, linux and macos.," GitHub, https://github.com/horsicq/Detect-It-Easy (accessed Nov. 19, 2024).

[5] "X64DBG," "x64dbg," https://x64dbg.com/ (accessed Nov. 19, 2024).

[6] R. Graham, No, a researcher didn't find Olympics app spying on you, https://blog.erratasec.com/2022/01/no-researcher-didnt-find-olympics-app.html (accessed Nov. 19, 2024).

[7] G. Hoglund, "Security band-aids: more cost-effective than 'secure' coding," IEEE Software, vol. 19, no. 6. Institute of Electrical and Electronics Engineers (IEEE), pp. 56, 58, Nov. 2002. doi: 10.1109/ms.2002.1049389. Available: http://dx.doi.org/10.1109/MS.2002.1049389