0.1 Project Description

Project 1: Given a crypto key and a crypto program crunching on it, write a tool that dumps all the instructions in the program that processes data that depends on the key. Write a one-page report on how the tool works and discuss the limitations.

0.2 Implementation

Two different approaches have been chosen to implement this project. The first one is a Dynamic Analysis approach while the second method is static. The dynamic analysis is done by using the HyperDbg debugger [1] and for the static analysis method, angr [2] is used.

0.2.1 Dynamic Analysis

For this approach, several pre-analysis steps have been done to make a working HyperDbg script. Multiple features [3, 4, 5, 6] can be used to trace the instructions and keys. Among them, I chose the '!monitor' [6] command as it is easier and faster because only access to the memory will create an event that can be used to gather key usages. This command employs the Extended Page Table (EPT) mechanism to monitor any activity (R/W/X) in the target address range (page) or in other words, in the key address range. The full description of this command is available here [7].

0.2.2 Intercepting System-calls

Firstly, we need to understand what is the system-call that is responsible for reading the key of the crypto program from the input. This way we could understand the address of where the key is stored in the memory.

After some investigations, I realized that *ntdll!NtReadFile* or system-call number **6** [8] is used for reading the input from the keyboard to the program.

By knowing the target system call, we could use the '!syscall' command [9] to intercept system-calls and once the target system-call is executed in the target program, we use the '!sysret' command [10] to intercept its results.

The definition of this system-call is available at MSDN [**?**]:

```
NTSTATUS NtReadFile(
  _In_     HANDLE           FileHandle,
  _In_opt_ HANDLE           Event,
  _In_opt_ PIO_APC_ROUTINE  ApcRoutine,
  _In_opt_ PVOID            ApcContext,
  _Out_    PIO_STATUS_BLOCK IoStatusBlock,
  _Out_    PVOID            Buffer,
  _In_     ULONG            Length,
  _In_opt_ PLARGE_INTEGER   ByteOffset,
  _In_opt_ PULONG           Key
);
```

As you can see, there is a *Buffer* field in the above function's definition. This parameter will later be filled with the target buffer (input of the keyboard).

As Windows uses the *fastcall* calling convention for x64 codes in both modules and system-calls, the parameters are passed in this order: *RCX*, *RDX*, *R8*, *R9*, then the rest of the parameters are on the stack starting from *RSP+28*, *RSP+30*, *RSP+38*, and so on. The *Buffer* is available at *RSP+30* as it's the fifth parameter. Please note that the *fastcall* calling convention order in Windows is different from Linux.

Once the input of the keyboard is filled (after returning from the SYSRET instruction [11]), we'll set a monitor on the target address using the following HyperDbg script (dslang [12]):

```
event c all
.start path "C:\Users\sina\Desktop\SimpleAesCrypt.exe"
g
g
print $pid
? .current_thread_id = $tid;
!sysret script {
  if ($tid == .current_thread_id) {
    .key_len =strlen(dq(@rsp+30));
    .key_start_addr = dq(@rsp+30);
    .key_end_addr = dq(@rsp+30) + .key_len;
    pause();
  }
}
g
g
event d all

output create MyOutputName1 file C:\Users\sina\Desktop\VUSec\
   accesses.txt
output open MyOutputName1
!monitor rw .key_start_addr .key_end_addr output {MyOutputName1
   } script {

  // printf("Access to: %x\n", $context);
  printf("%x\n", @rip & 0xffff);
}
g
output close MyOutputName1
```

After running the above script in HyperDbg, it saves the addresses of the memory where they tried to access the key buffer in a file named '**accesses.txt**'. At this point, the dynamic analysis phase is done and we could go to the next step.

0.2.3 Static Analysis

The above dynamic analysis is enough for finding all the instructions that try to access the key. However, if we want to see all the instructions in the program that process data that depends on the key, there might be other instructions that process the data without accessing the memory. For example, the program reads the key into the registers and using data in the registers, computes the encryption algorithm (e.g., AES).

The '!monitor' command is not able to report these cases. In order to solve this problem, we could use angr's DDG or Data Dependence Graph built on top of the CFG (Control Flow Graph).

In binary analysis, a Data Dependence Graph (DDG) is a representation that illustrates the dependencies between different data elements in a program or binary code. It is a graph-based data structure that helps analyze how data flows through a program and how different parts of the code interact with each other in terms of data dependencies.

The primary purpose of a Data Dependence Graph is to provide insights into the relationships and dependencies between variables, memory locations, and data operations within a program.

There are two primary concepts that we should consider here. First, the address that we read from the memory by using the dynamic analysis method is randomized by Windows ASLR (Address Space Layout Randomization), so we need to make sure that the base address in the memory matches with the base address in angr (by default, 0x400000 as it's a 32-bit program). For this purpose, we only gather the least 16 signification bits. There are also other ways to perform this conversion like reading the actual base address of the binary but 16 least significant bits are enough for this binary as it has a small code base.

0.3 Results

After running the DDG script, angr tries to find all the instructions that are dependent on the target instruction (that was proved to access the user input (key) from the previous dynamic analysis step). The below result shows the target instruction as well as dependent instructions.

Here is a portion of the results:

```
C:\Users\sina\Desktop\VUSec>python ddg.py
['0x1003', '0x1006', '0x1009', '0x100a', '0x100b', '0x100d', '0
   x100f', '0x1013', '0x1016', '0x101a', '0x101d', '0x1021', '0
   x1024', '0x1028', '0x102b', '0x102f', '0x1032', '0x1036', '0
   x1039', '0x103d', '0x1040', '0x1044']
0x400000
WARNING  | 2023-12-20 19:34:06,597 | angr.state_plugins.
   callstack | Returning to an unexpected address 0x7ffeffb4
```

```
WARNING  | 2023-12-20 19:34:07,919 | angr.state_plugins.
   callstack | Returning to an unexpected address 0x4
0x401920:       mov     cl, byte ptr [ebp + eax - 0x204]
0x40100b:       mov     esi, ecx
********************
0x40192a:       mov     byte ptr [ebp + eax - 0x405], cl
0x401006:       movzx   eax, byte ptr [edx]
********************
0x401920:       mov     cl, byte ptr [ebp + eax - 0x204]
0x40100b:       mov     esi, ecx
********************
0x40192a:       mov     byte ptr [ebp + eax - 0x405], cl
0x40100f:       movzx   eax, byte ptr [edx + 1]
********************

...
```

The complete list of results is available in a file named '**results.txt**'.

0.4 Limitations

This approach is pretty okay if the target program is not encrypted/packed/protected. Even though we can still extract instructions even if the program is packed, however, an anti-debug method might try to access the user-input buffer thousands of times in order to obfuscate the normal procedure. The other limitation is that the programmer might move the buffer several times in the memory and in these cases, the reverse engineer might use some manual investigation to find the address of the new buffers.

REFERENCES

[1] Mohammad Sina Karvandi, MohammadHosein Gholamrezaei, Saleh Khalaj Monfared, Soroush Meghdadizanjani, Behrooz Abbassi, Ali Amini, Reza Mortazavi, Saeid Gorgin, Dara Rahmati, and Michael Schwarz. Hyperdbg: Reinventing hardware-assisted debugging. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1709–1723, 2022.

[2] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[3] The HyperDbg Team. *i (instrumentation step-in) - HyperDbg Documentation*, 2023.

[4] The HyperDbg Team. *t (step-in) - HyperDbg Documentation*, 2023.

[5] The HyperDbg Team. *!track (track and map function calls and returns to the symbols) - HyperDbg Documentation*, 2023.

[6] The HyperDbg Team. *!monitor (monitor read/write/execute to a range of memory) - HyperDbg Documentation*, 2023.

[7] The HyperDbg Team. *Design of !monitor - HyperDbg Documentation*, 2023.

[8] J00ru. *Windows X86-64 System Call Table (XP/2003/Vista/2008/7/2012/8/10)*, 2023.

[9] The HyperDbg Team. *!syscall, !syscall2 (hook system-calls) - HyperDbg Documentation*, 2023.

[10] The HyperDbg Team. *!sysret, !sysret2 (hook SYSRET instruction execution) - HyperDbg Documentation*, 2023.

[11] The HyperDbg Team. *Design of !syscall !sysret - HyperDbg Documentation*, 2023.

[12] The HyperDbg Team. *Debugger Script (DS) - HyperDbg Documentation*, 2023.