An exercise in binary analysis and reverse compilation

KEYGENNING EVAL4

Nathan Rittenhouse – nathan_@mit.edu
Target

- http://crackmes.de/users/ezqui3l/eval_n4/browse
- Enter a user name and a serial number, the program says whether the input is valid or not
Our Purpose

- Understand eval4’s anti-debugging mechanisms
- Decompile eval4’s key verification algorithm
- Construct a program that when given a user name, will output a sequence of numbers that the program considers correct
Examining Input Flow

- Open target, attach
- Enter input, see results
- After dialog displays
  - Halt program
  - View call stack
- Observe
  - Program halted inside of MessageBoxA
Examining Input Flow

- Walk the call stack to find the DialogProc
- Examine the function
- Observe
  - Calls to GetDlgItemTextA, atoi, and sscanf
  - "%d" says the input should be decimal
  - A block of many mathematical operations
  - The hallmark of a key validation routine
Breaking The Algorithm

- Some global variables appear to be referenced
  - Always find out the global’s purpose if it feels relevant
  - IDA can easily show all direct references
- Examine the first global after the `sscanf`’s
  - Only one area of code modifies the value
  - It appears used inside of the long (~370 byte) arithmetic block
Anti-Debugging

- Inside the function referencing the global
  - FindWindowA with parameter “Ollydbg”
  - GetWindowThreadProcessId
- Follow the first non-symbolic call inside
  - Check references
  - This call is checksumming memory
- Clearly, the anti-debug mechanisms play a role in key validation
Anti-Debugging

- Close both olly and eval₄
  - Open olly, then eval₄
- Observe
  - Olly terminates
- To be sure the key validation mechanism is not affected by the debugger’s presence, we must reverse the anti-debugging before reversing the algorithm
A common culprit for anti-debugging code is a TLS callback
- These run when threads execute, which includes the time when a debugger is attached
- One is referenced by the first call in the anti-debug code calling block
Anti-Debugging

- Refer to the previous function called from `getOllydbgProcessId`.
- IDA shows this being used in a **huge** block of anti-debugging functions.

```asm
    call    killProgramIfItIsModified ; appears to be a block of checksumming functions
    call    scanAntiDebugFunctionsForBreakpoints
    call    decodeDisplayBoxAndGetSerialNumMethods
    call    scanStatusBoxAndGetSerialMethodsForBreakpoints
    push    3
    push    offset aRo0Geq1H ; "Ro0|Geq1H["H"
    call    scanAndKillEncodedProcessName
    push    1
    push    offset aPMmZCH/FYF ; "PMmZCH/FYF"
    call    scanAndKillEncodedProcessName
    push    2
    push    offset aUpF W2g2 ; "UpF W2g2"
    call    scanAndKillEncodedProcessName
    call    killProgramIfScanAndKillEncodedProcessNameIsModified
    call    scanGetOllydbgProcessIdForBPS
    call    getOllydbgProcessId
```
Implications

- We cannot
  - Set breakpoints
  - Use a program to debug that has “Ollydbg” as its title
  - Modify any **useful** functions without difficulty

- We **can**
  - Use hardware breakpoints
  - Patch Olly to remove the window naming
  - Statically analyze with impunity
Introduction to Decompilation

- Take assembly, recover high level meaning
- This involves
  - Reconstructing type information
    - In C, this is usually just length, signedness
  - Structure / array composition
  - Reconstructing program logic
    - Loops, control flow, arithmetic operations, etc.
**Type Recovery**

- **Size**
  - Look at register length
  - Look for `'[byte|word|dword] ptr`]
  - Then look for the sign extension

- **Sign**
  - `i` prefix – implies that the operation is signed
  - `idiv`/`div`
  - `imul`/`mul`
  - ‘Sign extended’ vs ‘zero extended’
  - `movsx`/`movzx`
Array Size

- Structure composition and array size must be revealed through contextual analysis
  - Note `var_6C` as the base of a signed integer array
    - `ecx*4` - an integer is 4 bytes in length
      - Each increment to ecx moves pointer to next array element
      - Ecx is incremented until it reaches 5
    - This decomposes to
      - `int var_6C[5]`
Loop Reconstruction

- Easy way
  - IDA can identify them
  - Watch for `cmp` then `jXX` opcode sequences to learn what conditions the loop must meet to continue

```
.text:004012F1  mov     edx, offset sub_401322
.text:004012F6  xor     ecx, ecx
.text:004012F8  cmp     edx, offset scanAntiDebugFunctionsForBreakpoint
.text:00401300  mov     ebp, esp
.text:00401302  jnb     short loc_401318
.text:00401302  loc_401302:  movzx    eax, byte ptr [edx]        ; CODE XREF: killProgramIfSemedi
.text:00401305  inc     edx
.text:00401306  add     ecx, eax
.text:00401308  cmp     edx, offset scanAntiDebugFunctionsForBreakpoint
.text:0040130E  jb      short loc_401302
.text:00401310  cmp     ecx, 0063Eh
.text:00401316  jz      short locret_40131F
.text:00401318  loc_401318:  ; CODE XREF: killProgramIfSemedi
```
Pattern #1 – Inline strlen

- [cl]ear [d]irection flag
  - Go forward, not backward in memory during the operation
- `repne scasb`
  - “While byte [edi] != al, decrement ecx by 1”
  - The not and dec turn ecx into the real string length
Pattern #2 – adc Optimizations

- [ad]d with [c]arry
- `cmp` is `sub` without actually changing register contents
  - This means `sub ecx, 1`
    - If ecx is zero, there will be a ‘carry’ because ecx will loop back to -1
    - If it loops, then unknown1 += 0 because the carry flag will be set
    - If not, unknown1 -= 1
Pattern #3 – sbb Optimizations

- [s]ubtract with [b]orrow
  - `adc` but with subtraction
- If eax == 0, carry flag is set
  - If not, carry flag is not set
- Translation
  - If carry flag set: `sub eax, (eax+1)`
    - Result: eax = -1
  - If carry flag not set: `sub eax, eax`
    - Result: eax = 0

```
mov   eax, ebx
and   eax, ecx
cmp   eax, 1
sbb   eax, eax
and   al, 20h
add   ecx, ecx
add   al, 41h
mov   [edx+ebp-0B8h], al
dec   edx
jns   short loc_402220
```
Decompiling the Monster

- Watch for writes to memory
  - In this case, to stack variables
- When the write is hit, observe operations on the value written before that point
  - Find the location where the variable was defined
    - Utilize IDA’s highlighting functionality
  - Watch for all operations to the variable after that point
  - Record for each variable involved in the write
int var_6c[5];
char var_4c[UNKNOWN];
char userNameCopy[UNKNOWN];

// the serial numbers are just sscan'd user supplied integers
var_A4 = (userNameCopy[0] * 2) - userNameCopy[5];
serialNumber2 = var_6c[1];
var_AC = (userNameCopy[0] * 2) - userNameCopy[7];
serialNumber1 = var_6c[0];
var_B4 = (userNameCopy[3] * 2) - userNameCopy[8];
var_BC = (userNameCopy[4] * 2) - userNameCopy[9];
var_C0 = serialNumber3 - var_6c[2];
serialNumber3 = var_C0;
var_C4 = serialNumber4 - var_6c[3];
serialNumber4 = var_C4;
var_C8 = serialNumber5 - var_6c[4];
serialNumber5 = var_C8;
var_CC = counter_getOllydbgProcessId;
var_E4 = counter_getOllydbgProcessId + counter_scanAndKillEncodedProcessName - 1;
var_E8 = var_E4 * 2 * 1;
var_4C = (var_A4 * var_E4 + serialNumber1) / var_E8;
var_4B = (serialNumber2 - var_6c[1]) + (userNameCopy[1] * 2 - userNameCopy[6]) * var_E4) / var_E8;
var_4A = ((var_AC * var_E4 + var_C0) / var_E8;
var_46 = (var_BC * var_E4 + var_C8) / var_E8;
var_47 = 0;
The intermediate result may be ugly
- Discern variables of interest
- Adjust decompilation to understand those variables
- From the disassembly, \texttt{var\_4C[x]} is the important range

```assembly
movzx   eax, [edx+ebp+var_4C]
cmp     byte ptr [edx+ebp+userName], al
setz    al
movzx   eax, al
add     [ebp+var_9C], eax
inc     edx
cmp     edx, 5
jnz     short loc_4019BB
cmp     [ebp+var_9C], 5 ; this seems to be the check that fails with the data
               ; I've inputted
jnz     short loc_4019F1
cmp     [ebp+var_CC], 3 ; this one, however, seems correct
jnz     short loc_4019F1
push    3              ; we want to hit here
```
var NC[0] = (((userNameCopy[9] * 2) - userNameCopy[5]) * (counter_GDBPId + counter_sAKEPN - 1)) * (serialNumber1 - var_6C[9]) / ((counter_GDBPId + counter_sAKEPN - 1) * 2 + 1);
var NC[1] = (((userNameCopy[1] * 2) - userNameCopy[6]) * (counter_GDBPId + counter_sAKEPN - 1)) * (serialNumber2 - var_6C[1]) / ((counter_GDBPId + counter_sAKEPN - 1) * 2 + 1);
var NC[2] = (((userNameCopy[2] * 2) - userNameCopy[7]) * (counter_GDBPId + counter_sAKEPN - 1)) * (serialNumber3 - var_6C[2]) / ((counter_GDBPId + counter_sAKEPN - 1) * 2 + 1);
var NC[3] = (((userNameCopy[3] * 2) - userNameCopy[8]) * (counter_GDBPId + counter_sAKEPN - 1)) * (serialNumber4 - var_6C[3]) / ((counter_GDBPId + counter_sAKEPN - 1) * 2 + 1);
var NC[4] = (((userNameCopy[4] * 2) - userNameCopy[9]) * (counter_GDBPId + counter_sAKEPN - 1)) * (serialNumber5 - var_6C[4]) / ((counter_GDBPId + counter_sAKEPN - 1) * 2 + 1);
var NC[5] = 0;
Debugger check variables are replaced with numbers present during a ‘normal’ run

```c
var_4C[0] = (((userNameCopy[0] * 2) - userNameCopy[5]) * 5 + (serialNumber1 - var_6C[0])) / 11;
var_4C[1] = (((userNameCopy[1] * 2) - userNameCopy[6]) * 5 + (serialNumber2 - var_6C[1])) / 11;
var_4C[2] = (((userNameCopy[2] * 2) - userNameCopy[7]) * 5 + (serialNumber3 - var_6C[2])) / 11;
var_4C[3] = (((userNameCopy[3] * 2) - userNameCopy[8]) * 5 + (serialNumber4 - var_6C[3])) / 11;
var_4C[4] = (((userNameCopy[4] * 2) - userNameCopy[9]) * 5 + (serialNumber5 - var_6C[4])) / 11;
var_4C[5] = 0;
```
Best

- This looks awfully like a loop

```c
for(i=8; i<5; i++)
    userName[i] = (((userNameCopy[i] * 2) - userNameCopy[i+5]) * 5 + (serialNumber[i] - var_6C[i])) / 11;
```
Solve The Equation

*Solve for the serial number*

```c
for(i=0; i<5; i++){
    serialNumber[i] = userName[i] * 11 - ((userNameCopy[i] * 2) - userNameCopy[i+5]) * 5 + var_6C[i];
    printf("Serial number part %d : %d\n", i, serialNumber[i]);
}
```
Game Over

```plaintext
C:\share>keygen_eval4.exe userName
Calculation 0 : 0x00000089
Calculation 1 : 0x00000079
Calculation 2 : 0x00000065
Calculation 3 : 0x0000006f
Calculation 4 : 0x00000029
Serial number part 0 : 739
Serial number part 1 : 781
Serial number part 2 : 707
Serial number part 3 : 810
Serial number part 4 : 694
Sanity check: userName

C:\share>keygen_eval4.exe nathan
Calculation 0 : 0x0000006e
Calculation 1 : 0x00000054
Calculation 2 : 0x00000087
Calculation 3 : 0x0000005c
Calculation 4 : 0x0000005a
Serial number part 0 : 770
Serial number part 1 : 731
Serial number part 2 : 736
Serial number part 3 : 776
Serial number part 4 : 707
Sanity check: nathan
```
Go Further

- Check out Rolf Rolle’s decompilation class
- Alex Sotirov’s speech on quirks with Microsoft specific code
- Opennce.org