• Introduction
• API Deobfuscation Method
  • Memory Access Analysis for Dynamic Obfuscation
  • Iterative Run-until API method for Static Obfuscation
• Implementation
• Demo
• Conclusion
Why API deobfuscation matters?

• Malwares hide functionalities by API obfuscation
  • Commercial packers obfuscate API functions
  • Malware authors have their own API obfuscator

• No deobfuscation tools for some modern packers
  • x64 packers
  • Custom packers
• Dynamic API Obfuscation
  • API functions are obfuscated during runtime
  • Instructions and addresses changes every run

Branch into a newly allocated block during execution time
(obfuscated User32.dll :MessageBox)
• Static API Obfuscation
  • API functions are obfuscated compile(packing) time
  • Instructions and addresses are the same

Branch into other section

API Call by ‘ret’ instruction
After deobfuscation, we have
  - (Near) original entry point
  - Recovered API function calls at OEP

With the deobfuscated image, we can do
  - Static analysis with disassembled and decompiled code
  - Dynamic analysis with debuggers
• How to deobfuscate API obfuscated binaries?
  • Dynamic API Obfuscation
    → Memory Access Analysis
  • Static API Obfuscation
    → Iterative Run-until-API Method

• How to evade anti-debugging?
  • Dynamic binary instrumentation (Intel Pin)
  • Anti-anti-debugger plugin in debuggers
  • Emulators
API Deobfuscation for Dynamic Obfuscators
• Memory Access Analysis
  • Relate *memory reads on API function code* and corresponding *memory writes on obfuscated code*
    • Instruction addresses of obfuscated API function → Original API function
  • Recover original API function by the obfuscated call target address
What happens during runtime obfuscation?
- Runtime obfuscator reads each function, obfuscates each instruction, writes the obfuscated code into a newly allocated memory
- Each function is obfuscated in sequence
• How can we identify the original API function?
  • Record every memory write before the next API function or DLL reads
  • Limit the number of memory write for the last API function
• Find OEP
  • Record every memory write and execute
  • OEP is the Last written address that is executed
  • Check written memory blocks (1 block = 4 Kbytes) to save memory
  • OEP is in the original executable file sections

Packed Section → Unpacked Section

Unpacked instruction is written
Packer

Additional Section

Unpack code is executed
Packer

Execution address Of written blocks
• Search for intermodular calls at OEP by pattern matching
  • Matched patterns may contain false positives
  • After target address resolution, misinterpreted instruction disappears
• Direct call resolution
  • If the call targets are in the constructed map from obfuscated addresses to API function, modify call targets to the original API function address
  • Generate a text file that contains resolved API function calls and OEP
• Indirect call resolution
  • Original segments (.text, .idata, ...) are merged into one segment by packing
  • Identify a memory block that contains successive obfuscated API function addresses
  • Modify obfuscated call addresses in the IAT candidate with the original API function
• Example: API Deobfuscation Information

Addresses are in RVA
• Generating a debugger script to resolve API calls
  • The text file generated by the memory access analyzer contains OEP, resolved obfuscated addresses
  • Implemented a python script to generate a debugger script that execute until OEP and resolve obfuscated addresses
• Debugging x86 binary with Ollydbg after running deobfuscation script
• Decompiled code with dumped file
• Static obfuscation pattern at OEP
  • Obfuscated call pattern
    • “Call qword ptr [___]” is changed into “Call rel32” when obfuscated
  • Obfuscated call run into API function
    • Stack shape is preserved
    • API call instruction and the first few instructions in the API function are obfuscated
    • After executing obfuscated instructions, execution reaches an instruction in the original API function
• Search obfuscated call by pattern
  • CALL rel32 – is a candidate
  • Check whether the address is in another section of the process

Call rel32; db 00
‘00’ after call break alignment
so that A few incorrect disassembled code occur

......
• Obfuscated code is executed until API function
• Run-until-API method
  • Change RIP into candidate API call address
  • Run until API function
• Integrity check
  • We need to check whether the stack pointer and the stack content is preserved after executing obfuscate call
Iterative run-until-API Method

• Apply run-until API method repeatedly on candidate obfuscated calls
  • Save context & Restore
Iterative run-until-API Method

- Iterative run-until-API method can be applied to various packers
  - VMP: API function call is virtualization-obfuscated
  - Themida64: API function call is mutated
  - Obsidium: The first few instructions in an API function are obfuscated
  - Custom packers
  - But, at last, execution is redirected into a real API function
Reversing with API Deobfuscator

- Debugging x64 binary with x64DBG after deobfuscation

Run correctly with resolved API call
Dumping x86/64 binary and static analysis with IDA Pro

API call recovered

IAT recovered
• Pin tool to resolve API Address
  • Windows 8.1/7 – 32/64 bit (on VMWare)
  • Visual Studio 2013
  • Intel Pin 2.14
• Python script to patch obfuscated call
• Reversing tools
  • X64dbg
  • IDA
Deobfuscation Process

API Resolver → API info

Debugger script Generator

dbg script → Debugging with x64dbg & Olly

dumped exe file → Static Analysis with IDA Pro
Reversing Packed Binary with API Deobfuscator

- Packed 32/64 bit samples
- Commercial packer packed 32bit malware
Conclusion
Conclusion

• Suggested two methods for API deobfuscation
  • Memory access analysis for dynamic obfuscation
  • Run-until-API method for static obfuscation

• Commercial packer protected binary can be analyzed using API deobfuscator
  • Using debugger
  • Using disassembler & decompiler
Limitation

- Depending on DBI tools
  - Packers can detect DBI tools
    - Defeating the transparency feature of DBI (BH US’14)
    - Ex) Obsidium detect Intel Pin as a debugger
  - DBI tools crash in some applications

- Static whole function obfuscated code cannot be deobfuscated
  - No instructions in the original API function is executed when the whole function is obfuscated
Future Work

• Anti-anti-debugging
  • Building x86/64 emulator for unpacking

• API function resolution
  • Code optimization and binary differencing for static whole function obfuscation
  • Backward dependence analysis for custom packers