txOne networks *The Leader of OT Zero Trust*

A New Trend for the Blue Team

Using a Practical Symbolic Engine to Detect Evasive Forms of Malware/Ransomware

@HITCON PEACE 2022

Sheng-Hao Ma
@aaaddress1

Mars Cheng
@marscheng_

Hank Chen
@hank0438

Who Are We?

Sheng-Hao Ma

Threat Researcher
PSIRT and Threat Research

- Spoke at Black Hat, DEFCON, HITB, VXCON, HITCON, ROOTCON, and CYBERSEC
- Instructor of CCoE Taiwan, Ministry of National Defense, Ministry of Education, and etc.
- The author of the popular security book "Windows APT Warfare: The Definitive Guide for Malware Researchers"

Mars Cheng

Manager
PSIRT and Threat Research

- Spoke at Black Hat, RSA Conference, DEFCON, SecTor, FIRST, HITB, ICS Cyber Security Conference, HITCON, SINCON, CYBERSEC, and CLOUDSEC
- Instructor of CCoE Taiwan, Ministry of National Defense, Ministry of Education, Ministry of Economic Affairs and etc.
- General Coordinator of HITCON 2022 and 2021
- Vice General Coordinator of HITCON 2020

Hank Chen

Threat Researcher
PSIRT and Threat Research

- Spoke at FIRST Conference in 2022
- Instructor of Ministry of National Defense
- Teaching assistant of Cryptography and Information Security Course in Taiwan NTHU and CCoE Taiwan
- Member of CTF team 10sec and ⚔TSJ⚔

txOne networks

Outline

- Introduction
  - Threat Overview
  - The Difficult Problem of Static/Dynamic Malware Detection and Classification
- Deep Dive into Our Practical Symbolic Engine
  - Related Work
  - Our Practical Symbolic Engine
- Demonstration
  - CRC32 & DLL ReflectiveLoader
  - Process Hollowing
  - Ransomware Detection
- Future Works and Closing Remarks

Outline

- **Introduction**
  - Threat Overview
  - The Difficult Problem of Static/Dynamic Malware Detection and Classification
- Deep Dive into Our Practical Symbolic Engine
  - Related Work
  - Our Practical Symbolic Engine
- Demonstration
  - CRC32 & DLL ReflectiveLoader
  - Process Hollowing
  - Ransomware Detection
- Future Works and Closing Remarks

Outline

Malware Type

Virus	Rootkit	Adware
Fileless Malware	Stealth Malware	Trojan
Malvertising	**Ransomware**	Spyware
Worms	Dropper	ShellCode

Threat Overview
Recent Attack Trends – Many Ransomware Family

Ransomware Family	2021 Q2	2021 Q3	2021 Q4	2022 Q1	From 2021 Q4 to 2022 Q1
WannaCry	62.38%	46.95%	46.73%	42.23%	↘
Cryptor	4.06%	17.72%	15.91%	13.79%	↘
Locker	10.44%	10.92%	10.57%	13.43%	↗
LockBit	2.10%	4.35%	5.32%	5.89%	↗
Conti	3.49%	3.09%	3.98%	4.34%	↗
Gandcrab	5.03%	5.21%	3.93%	4.19%	↗
Locky	5.59%	3.28%	3.32%	3.69%	↗
Cobra	2.61%	2.83%	2.73%	3.33%	↗
Hive	0.59%	0.79%	1.82%	2.56%	↗
MAZE	1.00%	1.27%	1.69%	2.07%	↗

txOne networks

The Ransomware Matrix

	WannaCry	Ryuk	Lockergoga	EKANS	RagnarLocker	ColdLock	Egregor	Conti v2
Language Check	No	No	No	No	Yes	No	Yes	No
Kill Process/Services	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Persistence	Yes	Yes	No	No	No	No	No	Yes
Privilege Escalation	Yes	Yes	No	No	Yes	No	No	No
Lateral Movement	Yes	No	No	No	No	No	No	No
Anti-Recovery	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Atomic-Check	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
File Encryption	R-M-W	R-W-M	M-R-W	R-W-M	R-W-M	R-W-M	R-W-M	R-W-M
Partial Encryption	No	Yes	No	No	No	Yes	Yes	Yes
Cipher Suite	AES-128-CBC RSA-2048	AES-256 RSA-2048	AES-128-CTR RSA-1024	AES-256-CTR RSA-2048	Salsa20 RSA-2048	AES-256-CBC RSA	ChaCha8 RSA-2048	ChaCha8 RSA-4096
Configuration File	Yes	No	No	Yes	Yes	No	Yes	No
Command-Line Arguments	Yes	No	Yes	No	Yes	No	Yes	Yes

File Encryption:
SF: SetFileInformationByHandle/NtSetInformationFile;
R: ReadFile ; W: WriteFile ; M: MoveFile; MP: MapViewOfFile, FF: FlushViewOfFile≈ç

Claim: The matrix is only based on the samples we had analyzed. They might add more features in their variants.

txOne networks

The Ransomware Matrix

	Bad Rabbit	Mount Locker	RansomExx	DoppelPaymer	Darkside	Babuk	REvil	LockBit 2.0
Language Check	No	No	No	No	Yes	No	Yes	Yes
Kill Process/Services	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Persistence	Yes	No	No	Yes	No	No	Yes	Yes
Privilege Escalation	Yes	No	No	Yes	No	No	Yes	Yes
Lateral Movement	Yes	Yes	No	No	No	No	No	Yes
Anti-Recovery	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Atomic-Check	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
File Encryption	MP-FF	R-W-SF	R-W-M	R-W-M	M-R-W	M-R-W	R-W-M	R-W-SF
Partial Encryption	Yes	Yes	No	No	Yes	Yes	Yes	Yes
Cipher Suite	AES-128-CBC RSA-2048	ChaCha20 RSA-2048	AES-256-ECB RSA-4096	AES-256-CBC RSA-2048	Salsa20 RSA-1024	HC256 Curve25519-ECDH	Salsa20 Curve255 19-ECDH	AES-128-CBC Curve25519-ECDH
Configuration File	No	No	No	No	Yes	No	Yes	No
Command-Line Arguments	Yes	Yes	No	No	Yes	Yes	Yes	Yes

File Encryption:
SF: SetFileInformationByHandle/NtSetInformationFile;
R: ReadFile ; W: WriteFile ; M: MoveFile; MP: MapViewOfFile, FF: FlushViewOfFile≈ç

Claim: The matrix is only based on the samples we had analyzed. They might add more features in their variants.

txOne networks

Malware detection Techniques

Type	Scope
Signature-based	Byte sequence, List of DLL, Assembly Instruction
Behavior-based	API Calls, System calls, CFG, Instruction trace, n-gram, Sandbox
Heuristic-based	API Calls, System call, CFG, Instruction trace, List of DLL, Hybrid featues, n-gram
Cloud-based	Strings, System calls, Hybrid featues, n-gram
Learning-based	API Calls, System call, Hybrid featues
…	

txOne networks

The Difficult Problem on Malware Detection

Type	Difficult Problem (Limitation)
Signature-based	Need huge database, Hard to defeat obfuscated samples, Vendor need to spend many people to update the signature
Behavior-based	Need to Run it, have the risk of attacking by 0-day exploits or vulnerabilities. Time-consuming and labor-intensive. Behavior policy can be bypassed
Heuristic-based	will include both of the above
Cloud-based	Immediacy of Internet connections. Adds additional delay to many tasks. Less effective at monitoring/detecting Heuristics
Learning-based	Learning dataset can't help to identify the variant
…	

txOne networks

The Difficult Problem on Malware Detection

- Time-consuming and labor-intensive when dynamic analysis
- Vendor need to update the signature based on different malware
- Can’t help to identify the variant
- Hard to defeat obfuscated samples

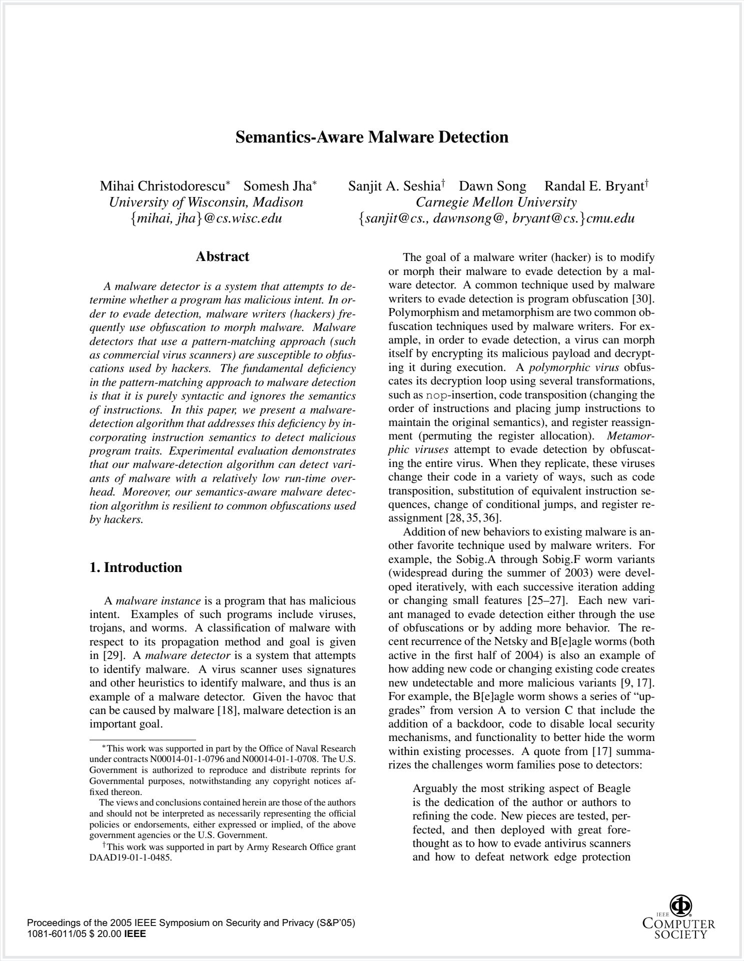
Outline

- Introduction
  - Threat Overview
  - The Difficult Problem of Static/Dynamic Malware Detection and Classification
- **Deep Dive into Our Practical Symbolic Engine**
  - Related Work
  - Our Practical Symbolic Engine
- Demonstration
  - CRC32 & DLL ReflectiveLoader
  - Process Hollowing
  - Ransomware Detection
- Future Works and Closing Remarks

Related Work

- Three main papers inspire us do this research
  - Christodorescu, Mihai, et al. "Semantics-aware malware detection." 2005 IEEE symposium on security and privacy (S&P'05). IEEE, 2005.
  - Kotov, Vadim, and Michael Wojnowicz. "Towards generic deobfuscation of windows API calls." arXiv preprint arXiv:1802.04466 (2018).
  - Ding, Steven HH, Benjamin CM Fung, and Philippe Charland. "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization." 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.
- Thanks for their contributions

Related Work



Semantics-Aware Malware Detection

Mihai Christodorescu* Somesh Jha*
University of Wisconsin, Madison
{mihai, jha}@cs.wisc.edu

Sanjit A. Seshia† Dawn Song Randal E. Bryant†
Carnegie Mellon University
{sanjit@cs., dawnsong@, bryant@cs.}cmu.edu

Abstract

A malware detector is a system that attempts to determine whether a program has malicious intent. In order to evade detection, malware writers (hackers) frequently use obfuscation to morph malware. Malware detectors that use a pattern-matching approach (such as commercial virus scanners) are susceptible to obfuscations used by hackers. The fundamental deficiency in the pattern-matching approach to malware detection is that it is purely syntactic and ignores the semantics of instructions. In this paper, we present a malware-detection algorithm that addresses this deficiency by incorporating instruction semantics to detect malicious program traits. Experimental evaluation demonstrates that our malware-detection algorithm can detect variants of malware with a relatively low run-time overhead. Moreover, our semantics-aware malware detection algorithm is resilient to common obfuscations used by hackers.

1. Introduction

A *malware instance* is a program that has malicious intent. Examples of such programs include viruses, trojans, and worms. A classification of malware with respect to its propagation method and goal is given in [29]. A *malware detector* is a system that attempts to identify malware. A virus scanner uses signatures and other heuristics to identify malware, and thus is an example of a malware detector. Given the havoc that can be caused by malware [18], malware detection is an important goal.

The goal of a malware writer (hacker) is to modify or morph their malware to evade detection by a malware detector. A common technique used by malware writers to evade detection is program obfuscation [30]. Polymorphism and metamorphism are two common obfuscation techniques used by malware writers. For example, in order to evade detection, a virus can morph itself by encrypting its malicious payload and decrypting it during execution. A *polymorphic virus* obfuscates its decryption loop using several transformations, such as `nop`-insertion, code transposition (changing the order of instructions and placing jump instructions to maintain the original semantics), and register reassignment (permuting the register allocation). *Metamorphic viruses* attempt to evade detection by obfuscating the entire virus. When they replicate, these viruses change their code in a variety of ways, such as code transposition, substitution of equivalent instruction sequences, change of conditional jumps, and register reassignment [28, 35, 36].

Addition of new behaviors to existing malware is another favorite technique used by malware writers. For example, the Sobig.A through Sobig.F worm variants (widespread during the summer of 2003) were developed iteratively, with each successive iteration adding or changing small features [25–27]. Each new variant managed to evade detection either through the use of obfuscations or by adding more behavior. The recent recurrence of the Netsky and B[e]agle worms (both active in the first half of 2004) is also an example of how adding new code or changing existing code creates new undetectable and more malicious variants [9, 17]. For example, the B[e]agle worm shows a series of "upgrades" from version A to version C that include the addition of a backdoor, code to disable local security mechanisms, and functionality to better hide the worm within existing processes. A quote from [17] summarizes the challenges worm families pose to detectors:

> Arguably the most striking aspect of Beagle is the dedication of the author or authors to refining the code. New pieces are tested, perfected, and then deployed with great forethought as to how to evade antivirus scanners and how to defeat network edge protection

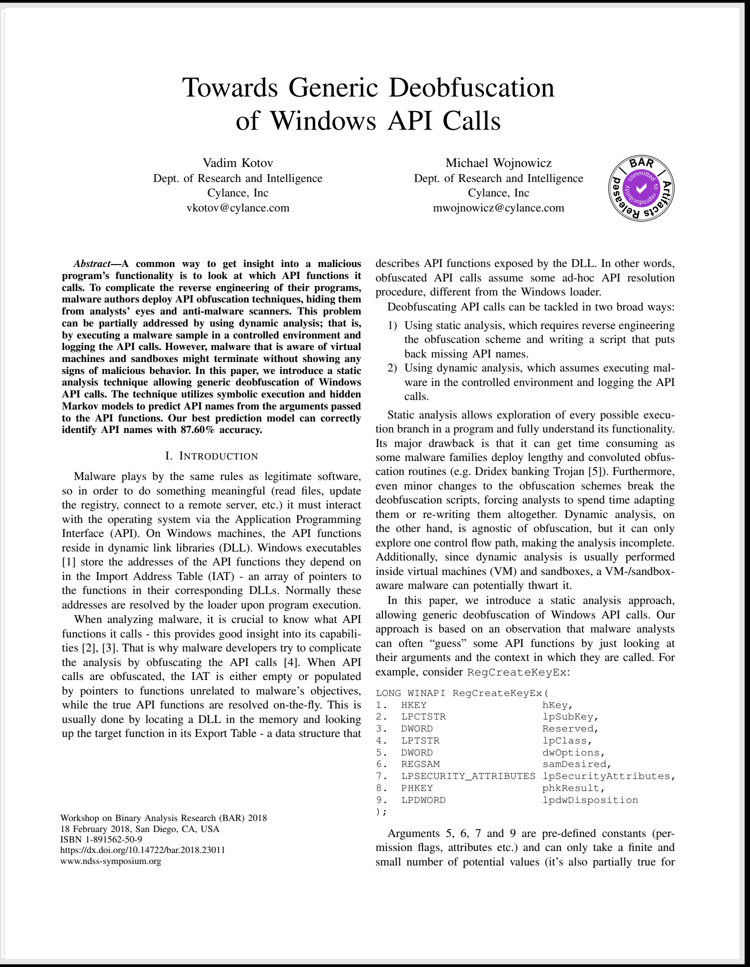
*This work was supported in part by the Office of Naval Research under contracts N00014-01-1-0796 and N00014-01-1-0708. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

†This work was supported in part by Army Research Office grant DAAD19-01-1-0485.

- Semantics-Aware Malware Detection (S&P'05)
- A lightweight malware template based on data reference relationships
- Efficient detection the same behavior but easily mutated code
- No False Positive!
- Nowadays: Practical Issues
  - The original paper only proposed the concept without releasing the engine and source code for use
  - Developing a complete symbolic engine to analyze real-world samples is difficult.
  - The Windows API recognition of strip symbols could not be resolved

Related Work



Towards Generic Deobfuscation of Windows API Calls

Vadim Kotov
Dept. of Research and Intelligence
Cylance, Inc
vkotov@cylance.com

Michael Wojnowicz
Dept. of Research and Intelligence
Cylance, Inc
mwojnowicz@cylance.com

***Abstract*—A common way to get insight into a malicious program's functionality is to look at which API functions it calls. To complicate the reverse engineering of their programs, malware authors deploy API obfuscation techniques, hiding them from analysts' eyes and anti-malware scanners. This problem can be partially addressed by using dynamic analysis; that is, by executing a malware sample in a controlled environment and logging the API calls. However, malware that is aware of virtual machines and sandboxes might terminate without showing any signs of malicious behavior. In this paper, we introduce a static analysis technique allowing generic deobfuscation of Windows API calls. The technique utilizes symbolic execution and hidden Markov models to predict API names from the arguments passed to the API functions. Our best prediction model can correctly identify API names with 87.60% accuracy.**

I. Introduction

Malware plays by the same rules as legitimate software, so in order to do something meaningful (read files, update the registry, connect to a remote server, etc.) it must interact with the operating system via the Application Programming Interface (API). On Windows machines, the API functions reside in dynamic link libraries (DLL). Windows executables [1] store the addresses of the API functions they depend on in the Import Address Table (IAT) - an array of pointers to the functions in their corresponding DLLs. Normally these addresses are resolved by the loader upon program execution.

When analyzing malware, it is crucial to know what API functions it calls - this provides good insight into its capabilities [2], [3]. That is why malware developers try to complicate the analysis by obfuscating the API calls [4]. When API calls are obfuscated, the IAT is either empty or populated by pointers to functions unrelated to malware's objectives, while the true API functions are resolved on-the-fly. This is usually done by locating a DLL in the memory and looking up the target function in its Export Table - a data structure that describes API functions exposed by the DLL. In other words, obfuscated API calls assume some ad-hoc API resolution procedure, different from the Windows loader.

Deobfuscating API calls can be tackled in two broad ways:

1) Using static analysis, which requires reverse engineering the obfuscation scheme and writing a script that puts back missing API names.
2) Using dynamic analysis, which assumes executing malware in the controlled environment and logging the API calls.

Static analysis allows exploration of every possible execution branch in a program and fully understand its functionality. Its major drawback is that it can get time consuming as some malware families deploy lengthy and convoluted obfuscation routines (e.g. Dridex banking Trojan [5]). Furthermore, even minor changes to the obfuscation schemes break the deobfuscation scripts, forcing analysts to spend time adapting them or re-writing them altogether. Dynamic analysis, on the other hand, is agnostic of obfuscation, but it can only explore one control flow path, making the analysis incomplete. Additionally, since dynamic analysis is usually performed inside virtual machines (VM) and sandboxes, a VM-/sandbox-aware malware can potentially thwart it.

In this paper, we introduce a static analysis approach, allowing generic deobfuscation of Windows API calls. Our approach is based on an observation that malware analysts can often "guess" some API functions by just looking at their arguments and the context in which they are called. For example, consider `RegCreateKeyEx`:

```
LONG WINAPI RegCreateKeyEx(
1.  HKEY                  hKey,
2.  LPCTSTR               lpSubKey,
3.  DWORD                 Reserved,
4.  LPTSTR                lpClass,
5.  DWORD                 dwOptions,
6.  REGSAM                samDesired,
7.  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
8.  PHKEY                 phkResult,
9.  LPDWORD               lpdwDisposition
);
```

Arguments 5, 6, 7 and 9 are pre-defined constants (permission flags, attributes etc.) and can only take a finite and small number of potential values (it's also partially true for

Workshop on Binary Analysis Research (BAR) 2018
18 February 2018, San Diego, CA, USA
ISBN 1-891562-50-9
https://dx.doi.org/10.14722/bar.2018.23011
www.ndss-symposium.org

- Towards Generic Deobfuscation of Windows API Calls (NDSS'18)
- Use Clever & Creative Ideas
  - Windows APIs are designed with many magic numbers that can be used as features for reverse engineering
    - For example, the RegCreateKeyExA parameter HKEY_CURRENT_USER evaluates to 0x80000001
  - Predict Windows API names by using only the parameter context distribution of function pointers
  - Using Hidden Markov Model (HMM): Up to 87.6% of API names can be recovered from the strip symbols binaries
- Practical Issues
  - Since the Markov Model is too rough in scale, APIs with less than four parameters cannot be analyzed
  - Not all API parameters have magic numbers used as features

Related Work

2019 IEEE Symposium on Security and Privacy

Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization

Steven H. H. Ding*, Benjamin C. M. Fung*, and Philippe Charland†
*Data Mining and Security Lab, School of Information Studies, McGill University, Montreal, Canada.
Emails: steven.h.ding@mail.mcgill.ca, ben.fung@mcgill.ca
†Mission Critical Cyber Security Section, Defence R&D Canada - Valcartier, Quebec, QC, Canada.
Email: philippe.charland@drdc-rddc.gc.ca

Abstract—**Reverse engineering is a manually intensive but necessary technique for understanding the inner workings of new malware, finding vulnerabilities in existing systems, and detecting patent infringements in released software. An assembly clone search engine facilitates the work of reverse engineers by identifying those duplicated or known parts. However, it is challenging to design a robust clone search engine, since there exist various compiler optimization options and code obfuscation techniques that make logically similar assembly functions appear to be very different.**

A practical clone search engine relies on a robust vector representation of assembly code. However, the existing clone search approaches, which rely on a manual feature engineering process to form a feature vector for an assembly function, fail to consider the relationships between features and identify those unique patterns that can statistically distinguish assembly functions. To address this problem, we propose to jointly learn the lexical semantic relationships and the vector representation of assembly functions based on assembly code. We have developed an assembly code representation learning model *Asm2Vec*. It only needs assembly code as input and does not require any prior knowledge such as the correct mapping between assembly functions. It can find and incorporate rich semantic relationships among tokens appearing in assembly code. We conduct extensive experiments and benchmark the learning model with state-of-the-art static and dynamic clone search approaches. We show that the learned representation is more robust and significantly outperforms existing methods against changes introduced by obfuscation and optimizations.

1. Introduction

Software developments mostly do not start from scratch. Due to the prevalent and commonly uncontrolled reuse of source code in the software development process [1], [2], [3], there exist a large number of clones in the underlying assembly code as well. An effective assembly clone search engine can significantly reduce the burden of the manual analysis process involved in reverse engineering. It addresses the information needs of a reverse engineer by taking advantage of existing massive binary data.

Assembly code clone search is emerging as an Information Retrieval (IR) technique that helps address security-related problems. It has been used for differing binaries to locate the changed parts [4], identifying known library functions such as encryption [5], searching for known programming bugs or zero-day vulnerabilities in existing software or Internet of Things (IoT) devices firmware [6], [7], as well as detecting software plagiarism or GNU license infringements when the source code is unavailable [8], [9]. However, designing an effective search engine is difficult, due to varieties of compiler optimizations and obfuscation techniques that make logically similar assembly functions appear to be dramatically different. Figure 1 shows an example. The optimized or obfuscated assembly function breaks control flow and basic block integrity. It is challenging to identify these semantically similar, but structurally and syntactically different assembly functions as clones.

Developing a clone search solution requires a robust vector representation of assembly code, by which one can measure the similarity between a query and the indexed functions. Based on the manually engineered features, relevant studies can be categorized into static or dynamic approaches. Dynamic approaches model the semantic similarity by dynamically analyzing the I/O behavior of assembly code [10], [11], [12], [13]. Static approaches model the similarity between assembly code by looking for their static differences with respect to the syntax or descriptive statistics [6], [7], [8], [14], [15], [16], [17], [18]. Static approaches are more scalable and provide better coverage than the dynamic approaches. Dynamic approaches are more robust against changes in syntax but less scalable. We identify two problems which can be mitigated to boost the semantic richness and robustness of static features. We show that by considering these two factors, a static approach can even achieve better performance than the state-of-the-art dynamic approaches.

P1: Existing state-of-the-art static approaches fail to consider the relationships among features. *LSH-S* [16], n-gram [8], n-perm [8], *BinClone* [15] and *Kam1n0* [17] model assembly code fragments as frequency values of operations and categorized operands. *Tracelet* [14] models assembly code as the editing distance between instruction sequences. *Discovre* [7] and *Genius* [6] construct descriptive features, such as the ratio of arithmetic assembly instructions, the number of transfer instructions, the number of basic blocks, among others. All these approaches assume each feature or category is an independent dimension. However, a *xmm0* Streaming SIMD Extensions (SSE) register is related to SSE operations such as *movaps*. A *fclose* libc function call is related to other file-related libc calls such as *fopen*. A *strcpy* libc call can be replaced with *memcpy*. These relationships provide more semantic information than

© 2019, Steven H. H. Ding. Under license to IEEE.
DOI 10.1109/SP.2019.00003

Authorized licensed use limited to: IEEE Xplore. Downloaded on April 01,2022 at 07:44:01 UTC from IEEE Xplore. Restrictions apply.

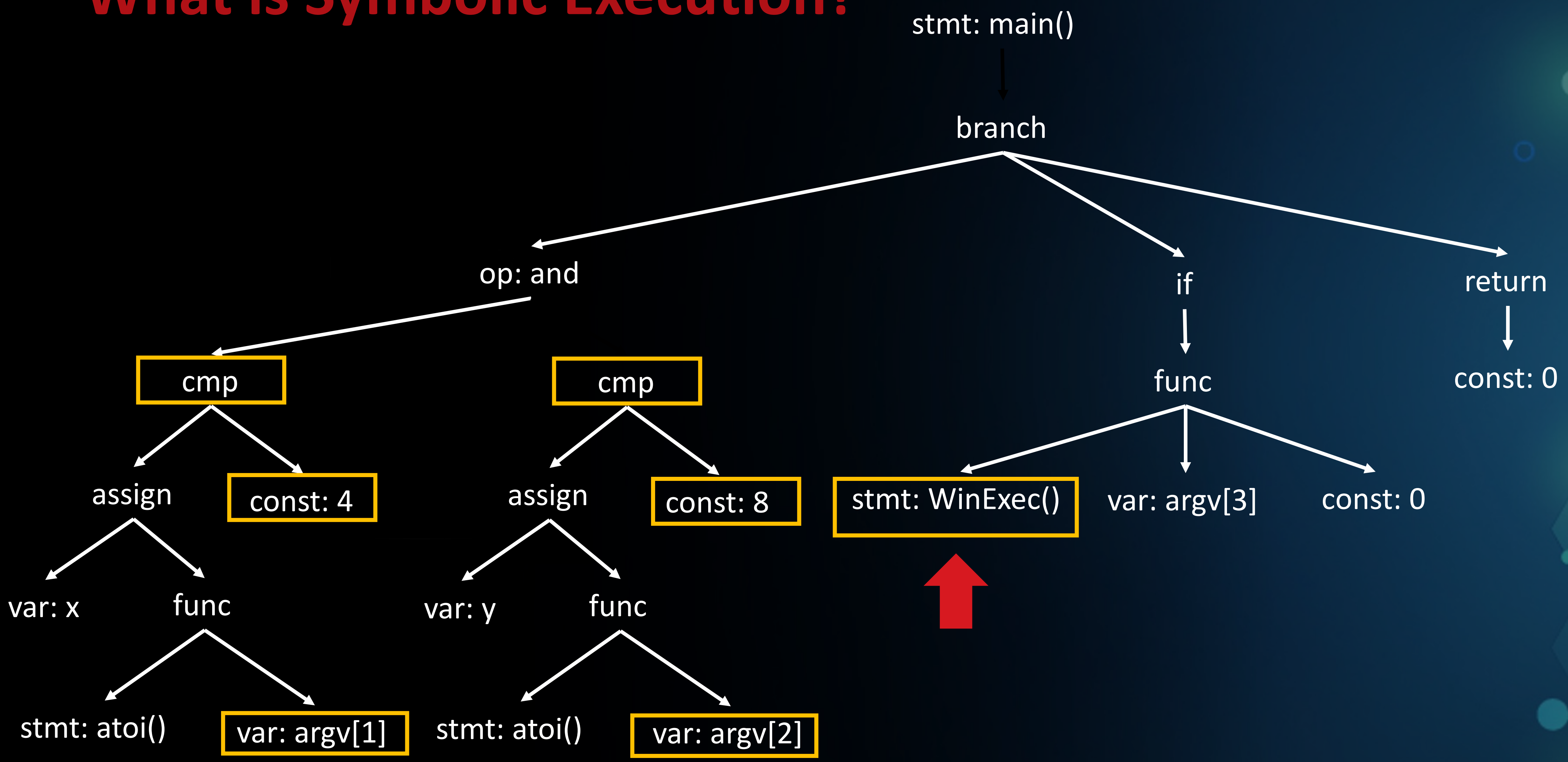
- Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization (S&P'19)
- Based on the Neural Network (NN) approach
  - Learn the instruction-level semantics of program binary effectively
  - Identify if an unknown binary is a variant of and similar to known programs
  - Even if OLLVM is fully enabled!
- Practical Issues
  - Non-explanatory: it is difficult to explain why this sample is identified as a known sample variant
  - Only works on classifying samples
  - Unable to precisely identify if binary has a specific malicious attack in a large number of behaviors

txOne networks

What is Symbolic Execution?

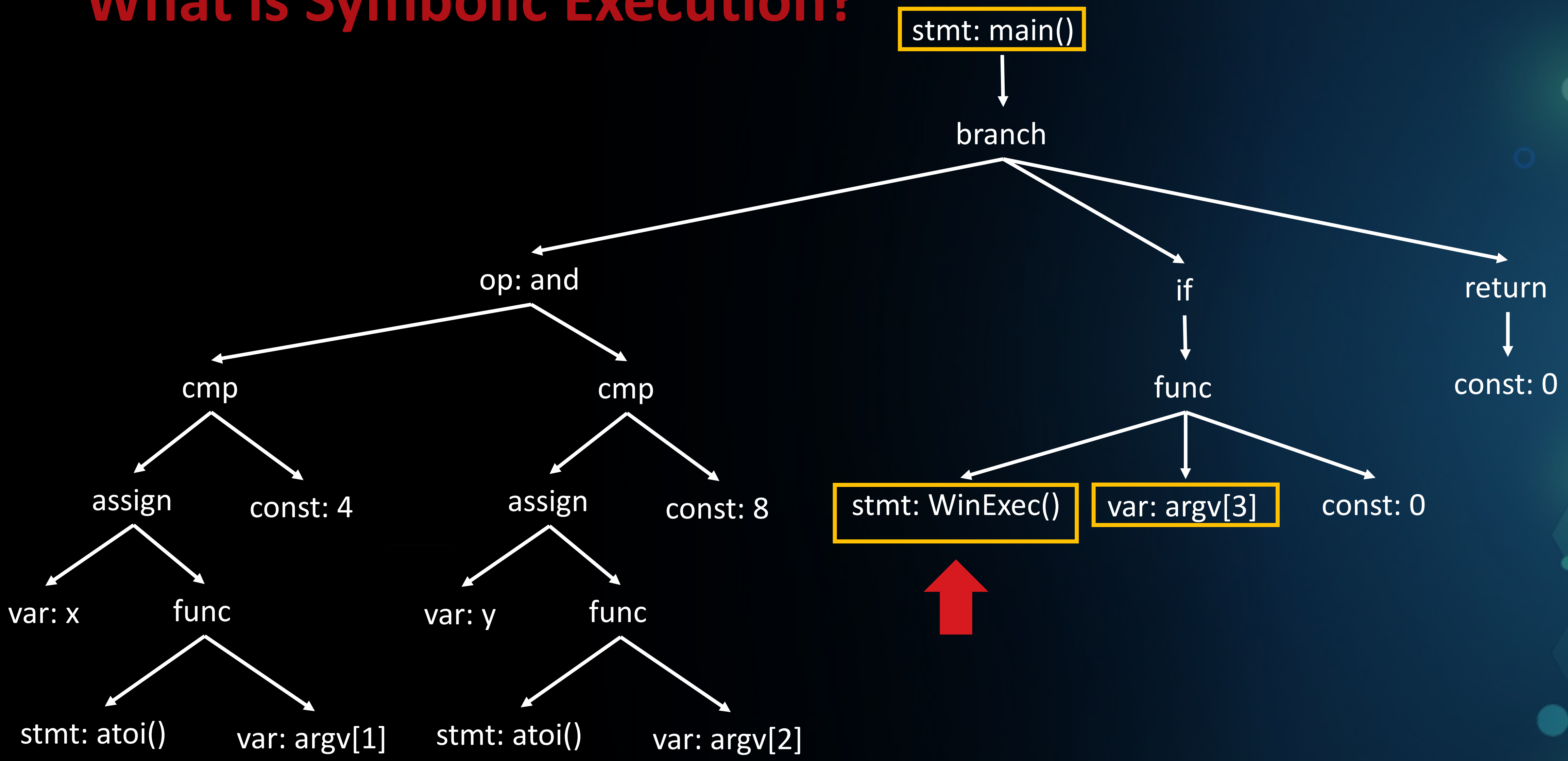
```c
#include <stdio.h>
#include <windows.h>

int main(int argc, char *argv[]){
    int x = atoi(argv[1]);
    int y = atoi(argv[2]);
    if ((x == 4) && (y == 8)){
        WinExec(argv[3], 0);
    }
    return 0;
}
```

What is Symbolic Execution?



txOne networks

What is Symbolic Execution?

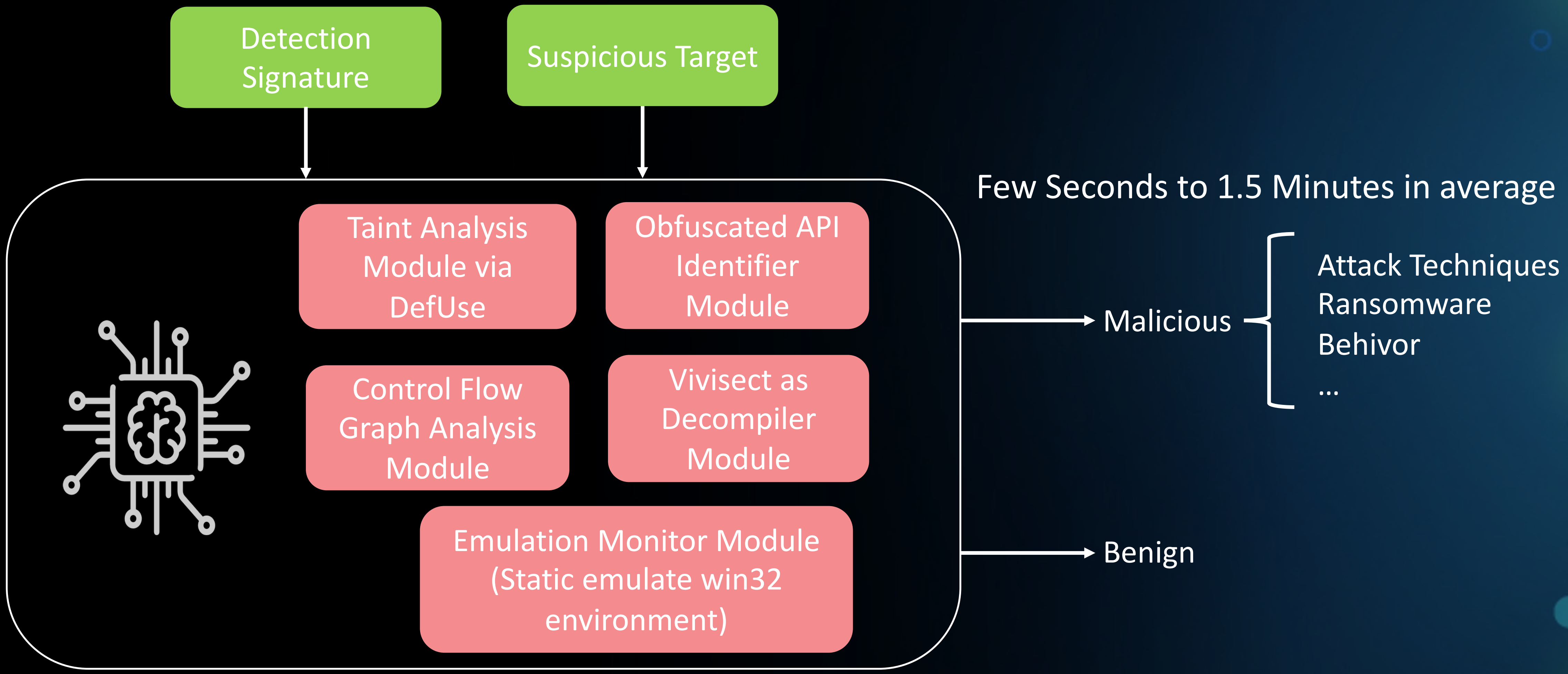


Why We Use Symbolic Execution to Solve Those Difficult Problem?

- Emulator: resource consumption, many problem about simulating environment, I/O, and can be bypassed
- Sandbox: Use real environment but also can be bypassed (Command line parameter, Anti-VM, Anti-sandbox, anti-debug…)
- Traditional Static analysis: can be bypassed easier. High false positives
- Symbolic Execution based: we use the lightweight part – DefUse relationship
  - It is enough to solve the problem of malware analysis, strengthen contextual relevance, semantic-based analysis, reduce false positives, and furthermore, full static analysis will not have the risk of being compromised
  - Low development cost and high adjustment flexibility

txOne networks

Our Practical Symbolic Engine

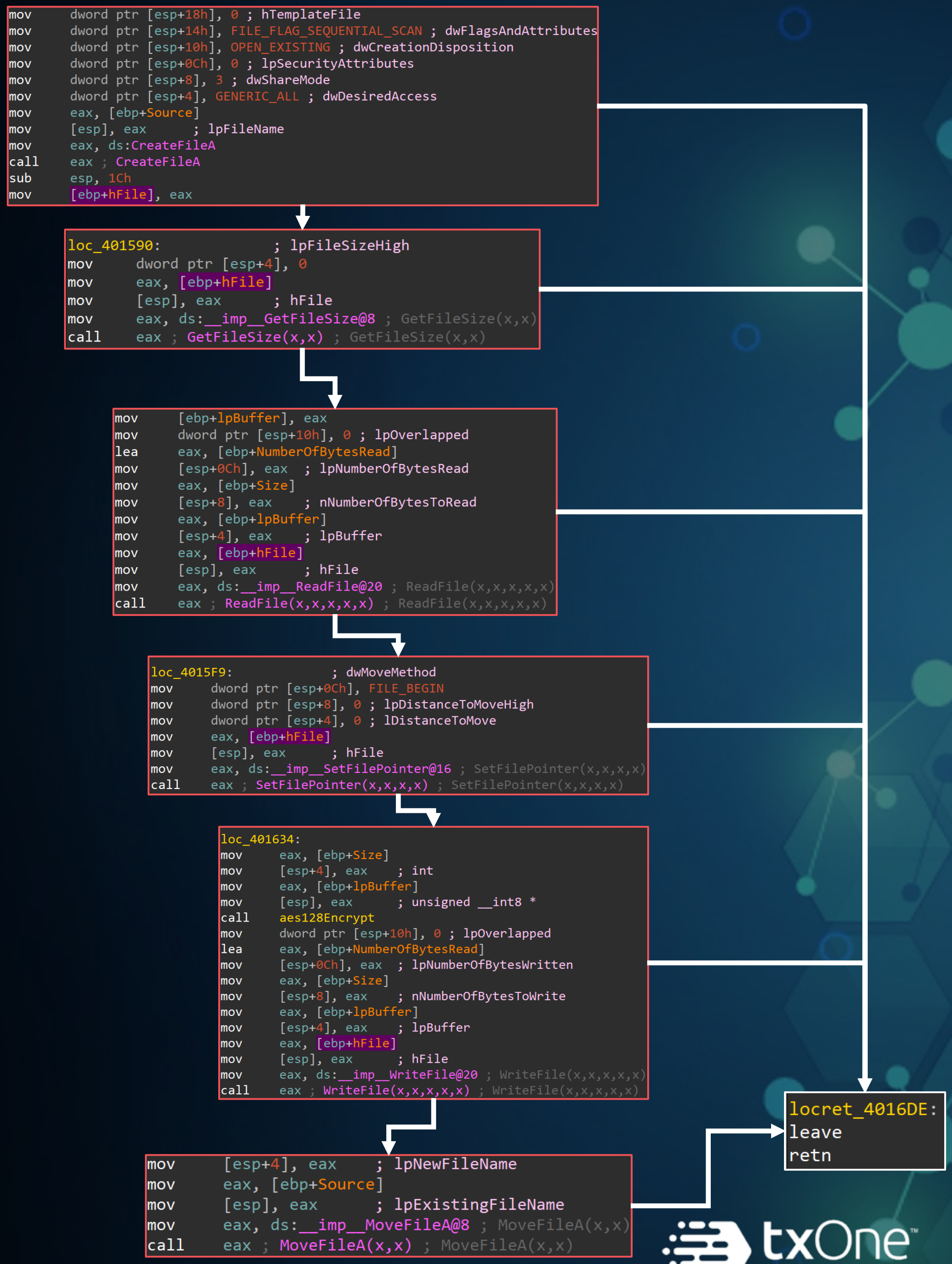


CFG Analysis Module

- Control Flow Graph (CFG) Analyze Module

```python
import viv_utils, IPython, sys, logging
logging.getLogger().setLevel(logging.CRITICAL)

if len(sys.argv) != 2:
    print(f'usage {sys.argv[0]} [file/to/scan]\n')
    sys.exit(0)

vw = viv_utils.getWorkspace(sys.argv[1], analyze=False, should_save=False)
vw.analyze()

def isAmbiguousRansomware(funcAddr):
    bool_OpenFile = bool_ReadFile = bool_fSeek = bool_WriteFile = False
    blocks = vw.getFunctionBlocks(funcAddr)
    for block in map(lambda b: BasicBlock(vw, *b), blocks):
        for ins in block.instructions():
            if vw.getComment(ins.va):
                print(vw.getComment(ins.va))
                bool_OpenFile  |= 'CreateFile' in vw.getComment(ins.va)
                bool_ReadFile  |= bool_OpenFile and ('ReadFile' in vw.getComment(ins.va))
                bool_fSeek     |= bool_ReadFile and 'SetFilePointer' in vw.getComment(ins.va)
                bool_WriteFile |= bool_fSeek and 'WriteFile' in vw.getComment(ins.va)
    return bool_OpenFile and bool_ReadFile and bool_fSeek and bool_WriteFile

for funcAddr in vw.getFunctions():
    if isAmbiguousRansomware(funcAddr):
        print(f'[+] found a function@{funcAddr:x} might be ransomware encrypt file.')
```

Parse function block based on our engine



txOne networks

Taint Analysis Module

- Taint Analysis Module via DefUse

Taint Analysis demo context result

```
exploit@exploit-lab: python3 ./TCSA/tcsa.py Sample/8257484c6d1a6dc94d6899e28b4da66e

TXOne Code Semantics Analyzer (TCSA) v1.
[OK] Rule Demo Attached.
0x401578 - kernel32.CreateFileA(0x4157100f, 0x10000000, 0x3, 0x0, 0x3, 0x8000000, 0x0) -> 0x4159b00f
0x4015a3 - kernel32.GetFileSize(0x4159b00f, 0x0) -> 0x415a100f
0x4015b1 - msvcrt.malloc(0x415a100f) -> 0x415a300f
0x4015e1 - kernel32.ReadFile(0x4159b00f, 0x415a300f, 0x415a100f, 0xbfb07f94, 0x0) -> 0x415a900f
0x40161c - kernel32.SetFilePointer(0x4159b00f, 0x0, 0x0, 0x0) -> 0x415af00f
0x401641 - sub_401520(0x415a300f, 0x415a100f, 0x0, 0x0, 0x0, 0xfefefefe) -> 0x415b100f
0x40166e - kernel32.WriteFile(0x4159b00f, 0x415a300f, 0x415a100f, 0xbfb07f94, 0x0) -> 0x415b700f
0x401693 - msvcrt.strcpy(0xbfb07e90, 0x4157100f) -> 0x415b900f
0x4016a9 - msvcrt.strrchr(0xbfb07e90, 0x2e) -> 0x415bb00f
0x4016d4 - kernel32.MoveFileA(0x4157100f, 0xbfb07e90) -> 0x415c100f
```

```python
def checkCall(self, starteip, endeip, op):

    if bool(op.iflags & envi.IF_CALL):
        rtype, rname, convname, callname, funcargs = self.getCallApi(endeip)
        callconv = self.getCallingConvention(convname)
        ...

        argv = callconv.getCallArgs(self, len(funcargs))
        ...

        # Windows API Hooks for Simulation e.g. msvcrt!sprintf()
        hook = self.hooks.get(callname)
        if ret is None and hook: hook(self, callconv, api, argv)

        elif self._func_only:
            if ret is None:
                ret = self.setVivTaint('apicall', (op, endeip, api, argv))
            retn = self.getProgramCounter()
            callconv.execCallReturn(self, ret, len(funcargs))
            ...
```

Part of Taint Analysis Example: all called APIs of static code, their return values are given by an assumed symbolic value, which can be used later to track the use of the situation.

Unknown API Recognition

- NDSS'18: Obfuscated API Identifier Module
- Real samples often have symbols removed or obfuscated, so fuzzy identification can help to identify what kind of API(s) it is, and thus determine what function it performs

```
op = self.emu.parseOpcode(starteip)
iscall = bool(op.iflags & envi.IF_CALL)
self.emu.op = op

# DefUse Case#1 - record data reference.
collect_reachDefinition(self, op, self.emu, starteip)

vg_path.getNodeProp(self.emu.curpath, 'valist').append(starteip)
endeip = self.emu.getProgramCounter()

# leak invoked call's arguments.
rtype, rname, convname, callname, funcargs = self.emu.getCallApi(endeip)
callname = f"sub_{endeip:x}" if callname == None else callname
callconv = self.emu.getCallingConvention(convname)

if len(funcargs) < 1 and ('sub_' in callname or callname == 'UnknownApi'):
    argv = callconv.getCallArgs(self.emu, 12) # dump max 12 stack values.
else:
    argv = callconv.getCallArgs(self.emu, len(funcargs))  # normal fetch argument info.
```

```
hFile = CreateFileA(lpFileName, 0x80000000, 1u, 0, 3u, 0x8000000u, 0);
if ( hFile == (HANDLE)-1 )
  return printf("Cannot open input file %s\n", lpFileName);
strcpy(&FileName, lpFileName);
pFileName = strrchr(&FileName, '.');
*(_DWORD *)pFileName = 0x6E61682E;
*((_DWORD *)pFileName + 1) = 0x6D6F7364;
*((_WORD *)pFileName + 4) = 0x65;
hObject = CreateFileA(&FileName, 0x40000000u, 0, 0, 2u, 0x80u, 0);
if ( hObject == (HANDLE)-1 )
  return printf("Cannot open output file!\n");
v30 = 0;
v29 = 0;
if ( _IAT_start__(&hProv, 0, L"Microsoft Enhanced RSA and AES Cryptographic Provider"
{
  v5 = GetLastError();
  printf("CryptAcquireContext failed: %x\n", v5);
  result = CryptReleaseContext(hProv, 0);
}
```

txOne networks

Prototype

```
[1096224783, 268435456, 3, 0, 3, 134217728, 0]
> special variables
> function variables
  0: 1096224783
  1: 268435456
  2: 3
  3: 0
  4: 3
  5: 134217728
  6: 0
  len(): 7
```

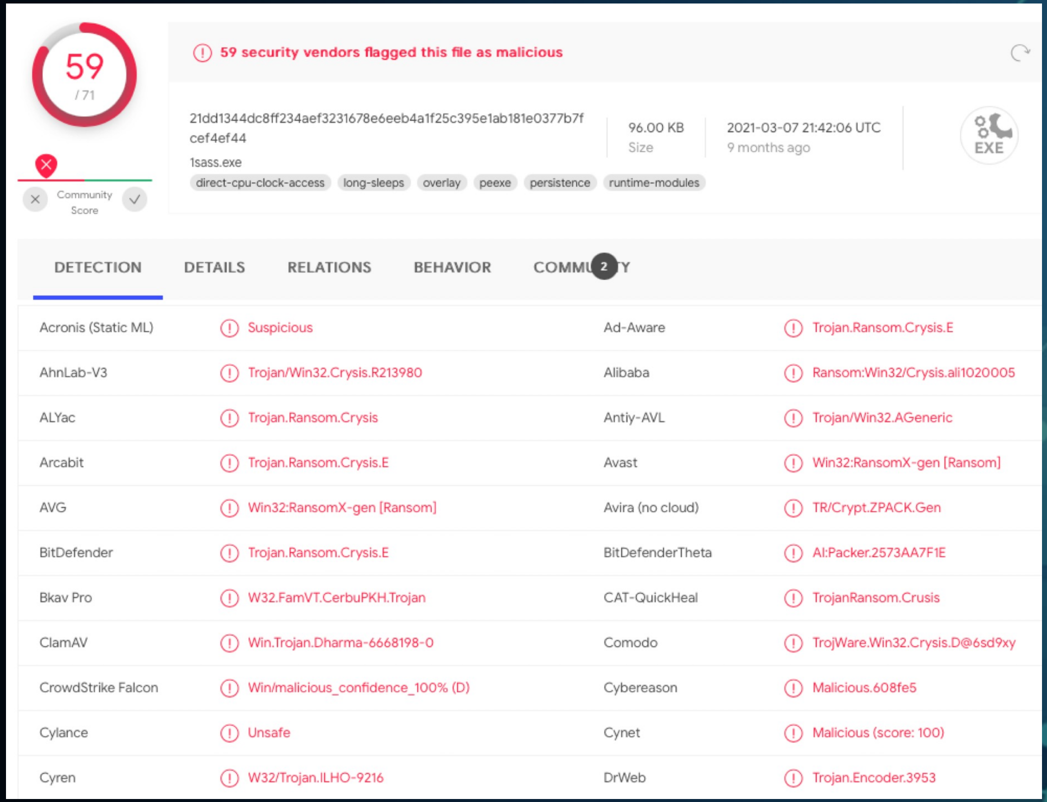
```
TCSA
TXOne Code Semantics Analyzer (TCSA) v1.
[OK] Rule Ransomware Attached.
kernel32.CreateFileA(0x4157100f, 0x10000000, 0x3, 0x0, 0x3, 0x8000000, 0x0)
```

```python
def verify_CreateFile(emu, starteip, op, iscall, callname, argv, ret):
    if ("CreateFileA" in callname) or ("CreateFileW" in callname)                                     \
        or ((len(argv) >= 7)                                                                        and \
        not isPointer(emu, argv[1]) and (argv[1] & 0xFFFFFFFF & (GENERIC_READ | GENERIC_WRITE | GENERIC_ALL))   and \
        not isPointer(emu, argv[2]) and (argv[2] == 0 or argv[2] & 0xFFFFFFFF & FILE_SHARE_GENERIC)        and \
        not isPointer(emu, argv[4]) and (argv[4] & 0xFFFFFFFF in (CREATE_ALWAYS, OPEN_EXISTING, CREATE_NEW, OPEN_ALWAYS)) and \
        not isPointer(emu, argv[5])):

        print(f"[v] CreateFileA found @ 0x{starteip:x} - {callname}({', '.join(hex(_) for _ in argv)}) -> {ret}")
```

txOne networks

Obfuscated Samples

- Obfuscated API Identifier Module
- Detect obfuscated ransomware samples
- Crysis
  - 21dd1344dc8ff234aef3231678e6eeb4a1f25c395e1ab181e0377b7fcef4ef44



Crysis

```
00409AF4 push    7FFFh
00409AF9 mov     eax, [ebp+arg_0]
00409AFC push    eax
00409AFD call    sub_407350
00409B02 add     esp, 8
00409B05 test    eax, eax
00409B07 jz      short loc_409B6C
```

```
00409B09 ; 8:     v6 = sub_406830();
00409B09 push    0
00409B0B push    0
00409B0D push    2
00409B0F push    0
00409B11 push    1
00409B13 push    40000000h
00409B18 mov     ecx, [ebp+arg_0]
00409B1B push    ecx
00409B1C call    sub_406830
00409B21 mov     [ebp+var_8], eax
00409B24 ; 9:     if ( v6 != -1 )
00409B24 cmp     [ebp+var_8], 0FFFFFFFFh
00409B28 jz      short loc_409B6C
```

```
00406830 sub_406830 proc near
00406830 mov     eax, ptrCreateFileA
00406835 jmp     eax
00406835 sub_406830 endp
```

```
BOOL __cdecl sub_409AE0(int a1, int a2, int a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+"

  v5 = 0;
  v7 = 0;
  if ( sub_407350(a1, 0x7FFF) )
  {
    v6 = sub_406830();
    if ( v6 != -1 )
    {
      v4 = sub_406910(v6, a2, a3, &v7, 0) && v7 == a3;
      v5 = v4;
      sub_406890(v6);
    }
  }
  return v5;
}
```

OLLVM - FLA (Obfuscation)

- Crysis

```
signed int __cdecl sub_4033B0(int a1, int a2, int a3, int a4, _DWORD *a5, unsigned
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v12 = 0;
  v13 = sub_406830();
  if ( v13 != -1 )
  {
    if ( sub_406720(v13, &v14) )
    {
      sub_406890(v13);
      if ( v14 )
      {
        if ( v14 <= 0x180000 )
          v12 = sub_402880(a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11);
        else
          v12 = sub_403090(a1, a2, a3, a4, (int)a5, a6, a7, a8, a9, a10, a11);
      }
    }
  }
  return v12;
}
```

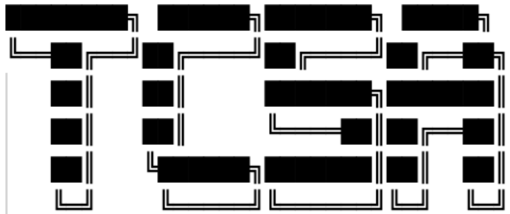
```
004033BD ; 7:    v13 = sub_406830();
004033BD push    0
004033BF push    0
004033C1 push    3
004033C3 push    0
004033C5 push    3
004033C7 push    80000000h
004033CC mov     eax, [ebp+arg_0]
004033CF push    eax
004033D0 call    sub_406830
004033D5 mov     [ebp+var_C], eax
004033D8 ; 8:    if ( v13 != -1 )
004033D8 cmp     [ebp+var_C], 0FFFFFFFFh
004033DC jz      loc_40348B
```

```
00406830 sub_406830 proc near
00406830 mov     eax, ptrCreateFileA
00406835 jmp     eax
00406835 sub_406830 endp
```

txOne networks

Engine Scan

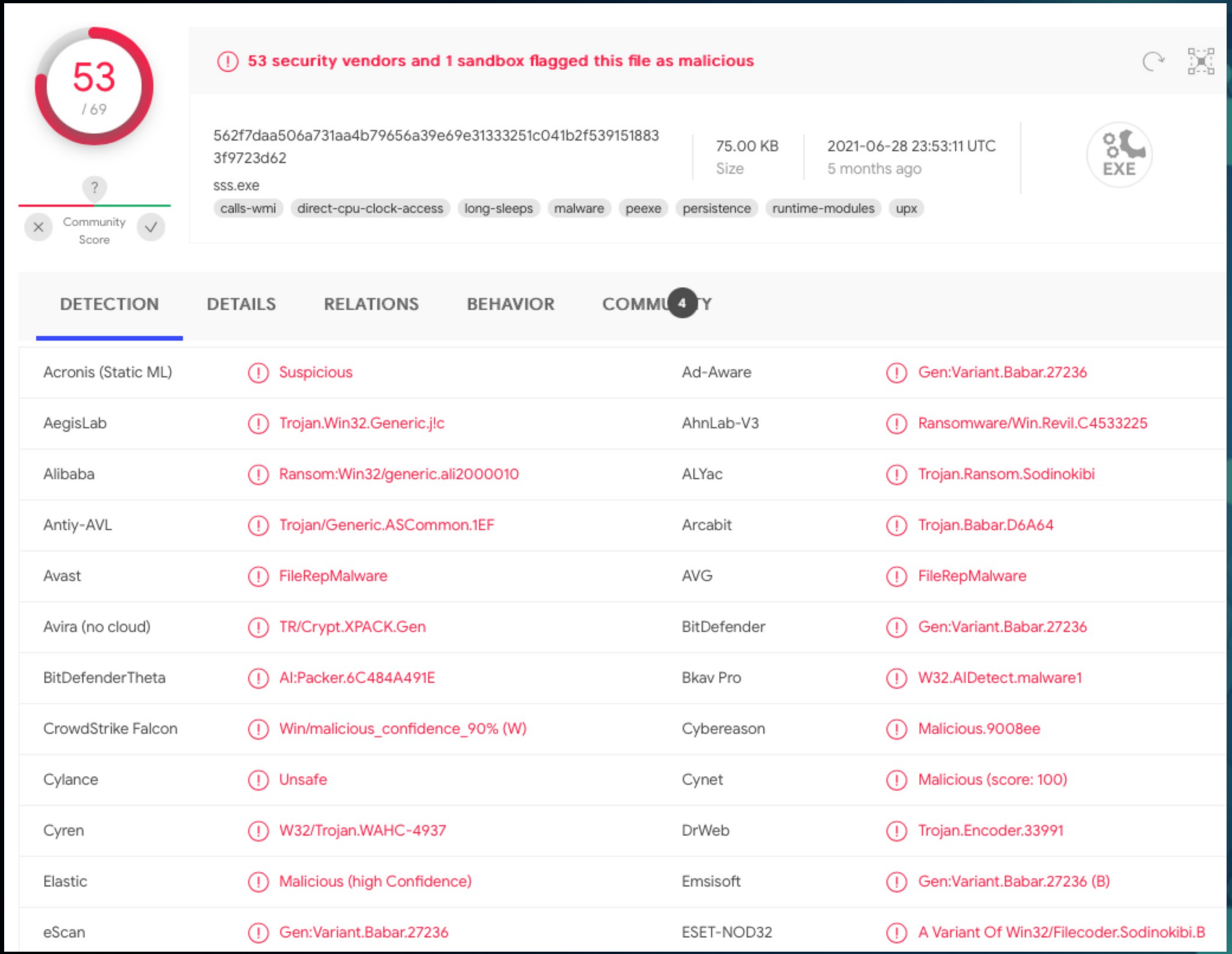
```
exploit@exploit-lab: python3 ./TCSA/tcsa.py Sample/21dd1344dc8ff234aef3231678e6eeb4a1f25c395e1ab181e0377b7fcef4ef44
```



```
TXOne Code Semantics Analyzer (TCSA) v1.
[OK] Rule Demo Attached.
[v] CreateFileA found @ 0x404d91 - UnknownApi(0x4156b00f, 0xfefe250b, 0x250b0822, 0x1, 0x1, 0x6161, 0x61616161) -> 1
[v] CreateFileA found @ 0x404d92 - UnknownApi(0x4156b00f, 0xfefe250b, 0x250b0822, 0x1, 0x1, 0x6161, 0x61616161) -> 1096224783
[v] CreateFileA found @ 0x404930 - UnknownApi(0x4156b00f, 0xfefefefe, 0xfefefefe, 0x0, 0x1, 0xfefe0002, 0x4159900f) -> 1096437775
[v] CreateFileA found @ 0x407768 - sub_406830(0x4157100f, 0x80000000, 0x1, 0x0, 0x3, 0x0, 0x0) -> 1096396815
[v] CreateFileA found @ 0x407776 - UnknownApi(0x4157100f, 0x80000000, 0x1, 0x0, 0x3, 0x0, 0x0) -> 1096396815
[v] CreateFileA found @ 0x407789 - sub_406830(0x4157300f, 0x40000000, 0x0, 0x0, 0x2, 0x0, 0x0) -> 1096421391
[v] CreateFileA found @ 0x4077e7 - UnknownApi(0x4157300f, 0x40000000, 0x0, 0x0, 0x2, 0x0, 0x0) -> 1096421391
[v] CreateFileA found @ 0x404931 - UnknownApi(0x4156b00f, 0xfefefefe, 0xfefefefe, 0x0, 0x1, 0xfefe0002, 0x4159900f) -> 1096437775
[v] CreateFileA found @ 0x404940 - UnknownApi(0x4156b00f, 0xfefefefe, 0xfefefefe, 0x0, 0x1, 0xfefe0002, 0x4159900f) -> 1096380431
[v] CreateFileA found @ 0x407797 - UnknownApi(0x4157300f, 0x40000000, 0x0, 0x0, 0x2, 0x0, 0x0) -> 1096421391
[v] CreateFileA found @ 0x407799 - UnknownApi(0x4157300f, 0x40000000, 0x0, 0x0, 0x2, 0x0, 0x0) -> 3216015268
[v] CreateFileA found @ 0x40539a - UnknownApi(0x4157500f, 0xfefefefe, 0xfefefefe, 0xfefefefe, 0x1, 0x0, 0x415a700f) -> 1096224783
[v] CreateFileA found @ 0x405369 - UnknownApi(0x61616161, 0xfefefefe, 0xfefefefe, 0xfefefefe, 0x1, 0x0, 0x415a700f) -> 1096224783
[v] CreateFileA found @ 0x409b1c - sub_406830(0x4157100f, 0x40000000, 0x1, 0x0, 0x2, 0x0, 0x0) -> 1096388623
[v] CreateFileA found @ 0x409b2a - UnknownApi(0x4157100f, 0x40000000, 0x1, 0x0, 0x2, 0x0, 0x0) -> 1096388623
[v] CreateFileA found @ 0x409b2c - UnknownApi(0x4157100f, 0x40000000, 0x1, 0x0, 0x2, 0x0, 0x0) -> 1096388623
[v] CreateFileA found @ 0x403164 - sub_406830(0x4157300f, 0xc0000000, 0x0, 0x0, 0x3, 0x0, 0x0) -> 1096429583
[v] CreateFileA found @ 0x403349 - UnknownApi(0x4157300f, 0xc0000000, 0x0, 0x0, 0x3, 0x0, 0x0) -> 1096429583
```

REvil

- 562f7daa506a731aa4b79656a39e69e31333251c041b2f5391518833f9723d62



REvil

- Obfuscated API Calls (GetProcAddress)

```
push    edi
push    80h
push    2
push    edi
push    edi
push    0C0000000h
push    [ebp+arg_8]
call    dword_412714
```

```
result = dword_412808(a4, a3, 0, *(v9 + 4), result, v9, 0);
if ( result )
{
  result = dword_412714(a5, 3221225472, 0, 0, 2, 128, 0);
  v13 = result;
  if ( result != -1 )
  {
    LOWORD(v21) = 19778;
    *(&v21 + 2) = v9[5] + 4 * v9[8] + 14 + *v9;
    v22 = 0;
    v23 = *v9 + 4 * v9[8] + 14;
    if ( dword_4125CC(result, &v21, 14, &v24, 0) && dword_41
    {
      if ( dword_4125CC(v13, v12, v9[5], &v24, 0) )
      {
        sub_405416(v13);
        result = dword_4127FC(v14, v12);
```

txOne networks

REvil

```
signed int __cdecl sub_4032A5(int a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+"

  v1 = 0;
  sub_406595(&unk_411278, 676, 12, 6, &v7);
  v8 = 0;
  v2 = sub_406A03(a1);
  v3 = sub_406A03(&v7);
  v4 = sub_405174(2 * (v3 + v2) + 2);
  sub_40695A(v4, a1);
  sub_406878(v4, &v7);
  v5 = dword_412714(v4, 0x40000000, 4, 0, 2, 67109120, 0);
  if ( v5 != -1 )
  {
    v1 = 1;
    CloseHandle(v5);
  }
  sub_4051C1(v4);
  return v1;
}
```

```
add     esp, 30h
push    edi             ; _DWORD
push    4000100h        ; _DWORD
push    2               ; _DWORD
push    edi             ; _DWORD
push    4               ; _DWORD
push    40000000h       ; _DWORD
push    esi             ; _DWORD
call    dword_412714
```

REvil

```
exploit@exploit-lab: python3 ./TCSA/tcsa.py Sample/562f7daa506a731aa4b79656a39e69e31333251c041b2f5391518833f9723d62

TXOne Code Semantics Analyzer (TCSA) v1.
[OK] Rule Demo Attached.
[v] CreateFileA found @ 0x404d4c - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 1096413199
[v] CreateFileA found @ 0x404d6e - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 1633771873
[v] CreateFileA found @ 0x404d70 - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 1633771873
[v] CreateFileA found @ 0x404d73 - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 1633771873
[v] CreateFileA found @ 0x404d76 - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 1212696660
[v] CreateFileA found @ 0x404d78 - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 1212696660
[v] CreateFileA found @ 0x404d7b - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 0
[v] CreateFileA found @ 0x404d7d - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 0
[v] CreateFileA found @ 0x404d80 - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 0
[v] CreateFileA found @ 0x404d83 - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 1633771873
[v] CreateFileA found @ 0x404d85 - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 3873892069
[v] CreateFileA found @ 0x404d88 - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 3873892083
[v] CreateFileA found @ 0x404d8b - UnknownApi(0x4157500f, 0xc0000000, 0x0, 0x0, 0x2, 0x80, 0x0) -> 3873892083
```

Deep Dive into Our Symbolic Engine

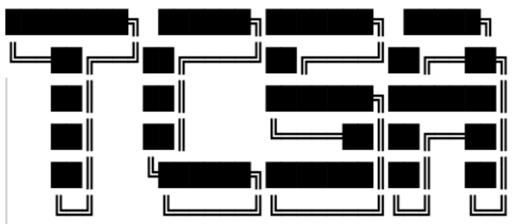
- TCSA (TXOne Code Semantics Analyzer)
  - Malware detection with instruction-level Semantic automata
  - Use Vivisect as the core decompiler engine
    - Support AMD, ARM, x86, MSP430, H8 and many other architectures
    - Support analysis of program files for Windows and Linux systems
  - Pure Python based Engine: Works on any platform able to run Python
  - In TCSA rule, developers can notate the data references between API calls
    - Symbolized return values of Win32 API, function, or unknown API
    - Usage of memory heap, stack, local variables, etc.
    - DefUse: tracing the source of data, memory values, argument values from
  - Support two additional feature extraction systems: YARA and Capa subsystems
  - Developers Orienting Malware Scanning Design
    - Developers can write their own Rules to be installed in the TCSA engine as callbacks
    - The TCSA engine will traverse and explore each function and the instructions in its Code Block
    - In the Callback, each instruction, memory, function name and parameter can be analyzed line by line

Deep Dive into Our Symbolic Engine

```python
def callback(emu, starteip, op, iscall, callname, argv, argv_snapshot, ret):
    if iscall:
        print(f"{starteip:x} - {op} ~ {callname}")
    else:
        print(f"{starteip:x} - {op}")
```

```c
int __cdecl main(int argc, char **argv)
{
  if ( argc == 2 && *(_DWORD *)argv[1] == 'nwP' )
    WinExec("cmd.exe", 1u);
  return 0;
}
```

```
exploit@exploit-lab: python3 ./TCSA/tcsa.py ~/Desktop/pwnExec.exe
```



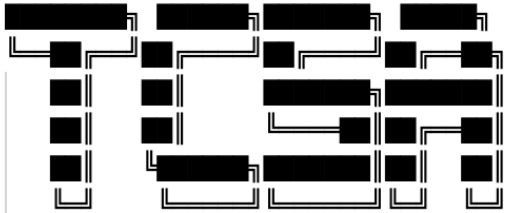
```
TXOne Code Semantics Analyzer (TCSA) v1.
[OK] Rule Demo Attached.
0x401000 - push ebp
0x401001 - mov ebp,esp
0x401003 - cmp dword [ebp + 8],2
0x401007 - jnz 0x00401024
0x401009 - mov eax,dword [ebp + 12]
0x40100c - mov eax,dword [eax + 4]
0x40100f - cmp dword [eax],0x006e7750
0x401015 - jnz 0x00401024
0x401017 - push 1
0x401019 - push 0x00402100
0x40101e - call dword [0x00402000] ~ kernel32.WinExec
 --- total used 2.596 sec ---
```

```
00401000 ; int __cdecl main(int argc, char **argv)
00401000 main proc near
00401000
00401000 argc= dword ptr  8
00401000 argv= dword ptr  0Ch
00401000 envp= dword ptr  10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 cmp     [ebp+argc], 2
00401007 jnz     short loc_401024

00401009 mov     eax, [ebp+argv]
0040100C mov     eax, [eax+4]
0040100F cmp     dword ptr [eax], 6E7750h
00401015 jnz     short loc_401024

00401017 push    1               ; uCmdShow
00401019 push    offset CmdLine  ; "cmd.exe"
0040101E call    ds:WinExec

00401024
00401024 loc_401024:
00401024 xor     eax, eax
00401026 pop     ebp
00401027 retn
```

Deep Dive into Our Symbolic Engine

```python
def callback(emu, starteip, op, iscall, callname, argv, argv_snapshot, ret):

    if iscall:
        argvlist = [ dumpMemory(emu, _) if isPointer(emu, _) else _  for _ in argv]
        print(f"{starteip:x} - {callname}{ tuple(argvlist) } -> {ret}")
```

```
exploit@exploit-lab: python3 TCSA/tcsa.py ~/Desktop/revShell.exe
```




```
TXOne Code Semantics Analyzer (TCSA) v1.
[OK] Rule Demo Attached.
0x140001025 - ws2_32.WSAStartup(514, 5368723024) -> 0x4159d00f
0x140001046 - ws2_32.WSASocketW(2, 1, 6, 0, 18374403896593350656, 18374403896593350656) -> 0x415a100f
0x14000105f - ws2_32.htons(4444) -> 0x415a500f
0x140001073 - ws2_32.inet_addr(b'127.0.0.1') -> 0x415a900f
0x1400010a3 - ws2_32.WSAConnect(1096421391, 5368723544, 16, 0, 0, 0) -> 0x415ad00f
0x140001150 - kernel32.CreateProcessA(0, b'cmd.exe', 0, 0, 1, 0, 0, 0, 5368723440, 5368722992) -> 0x415b100f
0x140001914 - kernel32.IsProcessorFeaturePresent(23) -> 0x4159d00f
 --- total used 2.951 sec ---
```

Deep Dive into Our Symbolic Engine

- Some functions that need to be implemented for the real Windows runtime results for pure static analysis
  - Process Execution Necessary: LoadLibrary, GetProcAddress, GetFullPathName, FindResource...
  - String handling Necessary: sprintf、scanf、lstrlenA...
  - Memory Handling Necessary: HeapAlloc、malloc、free...

```python
# msvcrt!sprintf() behavior
def msvcrt_sprintf(emu, callconv, api, argv):
    fmt = emu.readMemString(argv[1]).decode()

    stack_snapshot, stackArgvList = [], []
    for x in range(12): stack_snapshot.append( emu.readMemoryPtr(emu.getStackCounter() + 12 + x*4) )

    stackValIter = iter(stack_snapshot)
    for eachFmt in re.findall(r"\%[diouXxfFeEgGaAcSsbn$.]",fmt):
        if eachFmt[-1] == 's' or eachFmt[-1] == 'S':
            # cache max 32 alphabets for wstring-like api name.
            bytearrApiName = emu.readMemory(next(stackValIter), 64)
            stackArgvList.append(bytearrApiName.decode('utf-16' if eachFmt[-1] == 'S' else 'utf-8').split('\x00')[0])
        if eachFmt[-1] in 'di' or eachFmt[-1] in 'DI':
            stackArgvList.append(next(stackValIter))

    szAnser = (fmt.replace('%S', '%s') % tuple(stackArgvList)).encode() + b'\x00'
    emu.writeMemory(argv[0], szAnser)
    callconv.execCallReturn(emu, 0xdeadbeef, len(argv)) # return value of sprintf is useless.
```

txOne networks

Deep Dive into Our Symbolic Engine

```python
def callback(emu, starteip, op, iscall, callname, argv, stacksnapshot, ret):
    # receive each instruction from TCSA engine
    # here we can read memory, stack, api name, arguments, ...
    pass

def initialize( In_chakraCore ):
    global chakraCore
    chakraCore = In_chakraCore
    print('[OK] Rule Demo Attached.')
    # ...

def cleanup( In_chakraCore, In_capaMatchRet, In_yaraMatchRet ):
    # ...
```

- Malware Rule/Automata Developing
  - Each TCSA Rule should have at least three callback, initialize, and cleanup callback functions.
  - In the initialize function, developers have the ability to do some necessary preparation
  - Developers can receive each instruction in the callback function with execution status from the TCSA engine
    - Used to extract and collect instruction level features to identify specific behavior in a function
    - Locate and mark potentially suspicious function
  - Developers can make the final decision in the cleanup function to determine if a specific behavior has been found
    - Based on the features collected in the callback
    - based on the YARA/CAPA Rule match features

txOne networks

Outline

- Introduction
  - Threat Overview
  - The Difficult Problem of Static/Dynamic Malware Detection and Classification
- Deep Dive into Our Practical Symbolic Engine
  - Related Work
  - Our Practical Symbolic Engine
- **Demonstration**
  - CRC32 & DLL ReflectiveLoader
  - Process Hollowing
  - Ransomware Detection
- Future Works and Closing Remarks

txOne networks

CRC32

```python
def callback(emu, starteip, op, iscall, callname, argv, argv_snapshot, ret):

    if not hasattr(callback, "gc") or callback.gc['currFunc'] != emu.funcva:
        callback.gc = {
            'currFunc' : emu.funcva, 'magic' : False,
            'loop8' : False, 'xor' : False, 'detect' : False
        }

    # collect features from the assembly code
    argValues = [ emu.getOperValue(op, _) for _ in range(len(op.opers)) ]

    # crc32 magic
    if 0xEDB88320 in argValues: callback.gc['magic'] = True

    # should loop for 8 times
    if 8 in argValues: callback.gc['loop8'] = True

    # use xor
    if 'xor' in op.mnem: callback.gc['xor'] = True

    if iscall and 'RtlComputeCrc32' in callname:
        print(f"[v] found CRC32 at sub_{callback.gc['currFunc']:x} - by ntdll!RtlComputeCrc32")
        callback.gc['detect'] = True

    if callback.gc['magic'] and callback.gc['loop8'] and callback.gc['xor']:
        if not callback.gc['detect']:
            print(f"[v] found CRC32 at sub_{callback.gc['currFunc']:x} - by Binary Feature")
            callback.gc['detect'] = True
```

```c
unsigned int crc32b(unsigned char *message) {
    unsigned int byte, crc, mask;
    int i, j;

    i = 0;
    crc = 0xFFFFFFFF;
    while (message[i] != 0) {
        byte = message[i];            // Get next byte.
        crc = crc ^ byte;
        for (j = 8; j > 0; j--) {    // Do eight times.
            mask = -(crc & 1);
            crc = (crc >> 1) ^ (0xEDB88320 & mask);
        }
        i = i + 1;
    }
    return ~crc;
}
```

txOne networks

CRC32 (Cont.)

```
LPVOID __stdcall sub_403CBD(LPVOID lpParameter)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v1 = lpParameter;
  *(lpParameter + 2) = 0;
  v2 = sub_404AB4();
  *(lpParameter + 1) = v2;
  *v2 = v2;
  *(v1[1] + 4) = v1[1];
  *(v1[1] + 8) = v1[1];
  *(*(lpParameter + 1) + 20) = 1;
  *(*(lpParameter + 1) + 21) = 1;
  v9 = 0;
  *(lpParameter + 10) = 0;
  *(lpParameter + 11) = 0;
  *(lpParameter + 12) = 0;
  LOBYTE(v9) = 1;
  sub_40389A();
  v3 = 0;
  v4 = (lpParameter + 96);
  do
  {
    v5 = v3;
    v6 = 8;
    do
    {
      if ( v5 & 1 )
        v5 = (v5 >> 1) ^ 0xEDB88320;            // CRC32 Loop
      else
        v5 >>= 1;
      --v6;
    }
    while ( v6 > 0 );
    *v4 = v5;
    ++v3;
    ++v4;
  }
  while ( v3 < 256 );
```

```
exploit@exploit-lab: python3 ./TCSA/tcsa.py Sample/
e606530456555bfa92c98365539b16d20bf678fae1ce180d9574a0ea48cc8a9f

TXOne Code Semantics Analyzer (TCSA) v1.
[OK] Rule CRC32 Attached.
[v] found CRC32 at sub_403cbd - by Binary Feature
```

ReflectiveLoader

```
// src: https://github.com/stephenfewer/ReflectiveDLLInjection
DLLEXPORT ULONG_PTR WINAPI ReflectiveLoader( VOID ) {
    ...

    // STEP 2: load our image into a new permanent location in memory...
    // get the VA of the NT Header for the PE to be loaded
    uiHeaderValue = PIMAGE_NT_HEADERS(uiLibraryAddress + ((PIMAGE_DOS_HEADER)uiLibraryAddress)->e_lfanew);

    // allocate all the memory for the DLL to be loaded into. we can load at any address because we will
    // relocate the image. Also zeros all memory and marks it as READ, WRITE and EXECUTE to avoid any problems.
    uiBaseAddress = (ULONG_PTR)pVirtualAlloc(
        NULL, uiHeaderValue->OptionalHeader.SizeOfImage, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE );

    // we must now copy over the headers
    ...

    // STEP 3: load in all of our sections...
    // File Mapping: iterate through all sections, loading them into memory.
    uiValueA = ( (ULONG_PTR)&uiHeaderValue->OptionalHeader + uiHeaderValue->FileHeader.SizeOfOptionalHeader );
    ...

    // STEP 4: process our images import table...
    uiValueB = (ULONG_PTR)&uiHeaderValue->OptionalHeader.DataDirectory[ IMAGE_DIRECTORY_ENTRY_IMPORT ];
    uiValueC = ( uiBaseAddress + ((PIMAGE_DATA_DIRECTORY)uiValueB)->VirtualAddress );
    ...

    // STEP 6: call our images entry point
    // uiValueA = the VA of our newly loaded DLL/EXE's entry point
    uiValueA = ( uiBaseAddress + uiHeaderValue->OptionalHeader.AddressOfEntryPoint );
    ...

    // call our respective entry point, fudging our hInstance value
    ((DLLMAIN)uiValueA)( (HINSTANCE)uiBaseAddress, DLL_PROCESS_ATTACH, lpParameter );
```

- Traversing memory to locate its own PE Image address
- Parsing its own IMAGE_NT_HEADERS structure
  - Allocate the memory of the OptionalHeader.SizeOfImage size using VirtualAlloc.
  - Mapping each section to its own PE Image to this new memory
  - Parse OptionalHeader.DataDirectory to resolve and repair the import table
  - Parse OptionalHeader.AddressOfEntryPoint and call entry

ReflectiveLoader (Cont.)

```python
def callback(emu, starteip, op, iscall, callname, argv, argv_snapshot, ret):
    if not hasattr(callback, "gc") or callback.gc['currFunc'] != emu.funcva:
        callback.gc = { 'currFunc' : emu.funcva,
            'ntHdrList' : list(), 'sizeOfImgList' : list(), 'impAddrDrList' : list(), 'entryAddrList' : list(),
            'newImageAt' : 0xffffffff, 'entryRva' : 0xffffffff, 'detect' : False }

    argValues = [ emu.getOperValue(op, _) for _ in range(len(op.opers)) ]
    if op.mnem == 'cmp' and 0x4550 in argValues: # try to parse "PE" field?
        for _ in range(len(op.opers)):
            if guessNtHdrPtr := emu.getOperAddr(op, _):
                # append this guess ntHdr addr into watch list.
                callback.gc['ntHdrList'].append(guessNtHdrPtr)

                # append the value of ntHdr.sizeOfImg into watch list.
                callback.gc['sizeOfImgList'].append(emu.readMemoryPtr(guessNtHdrPtr + 0x50))

                # append the address of ntHdr.DataDir[IMPORT_DIR] into watch list.
                callback.gc['impAddrDrList'].append(emu.readMemoryPtr(guessNtHdrPtr + 0x80))

                # append the ntHdr.AddressOfEntry into watch list.
                callback.gc['entryAddrList'].append(emu.readMemoryPtr(guessNtHdrPtr + 0x28))
                print(f"[*] found NtHdr parsing on {starteip:x} - {op}")
```

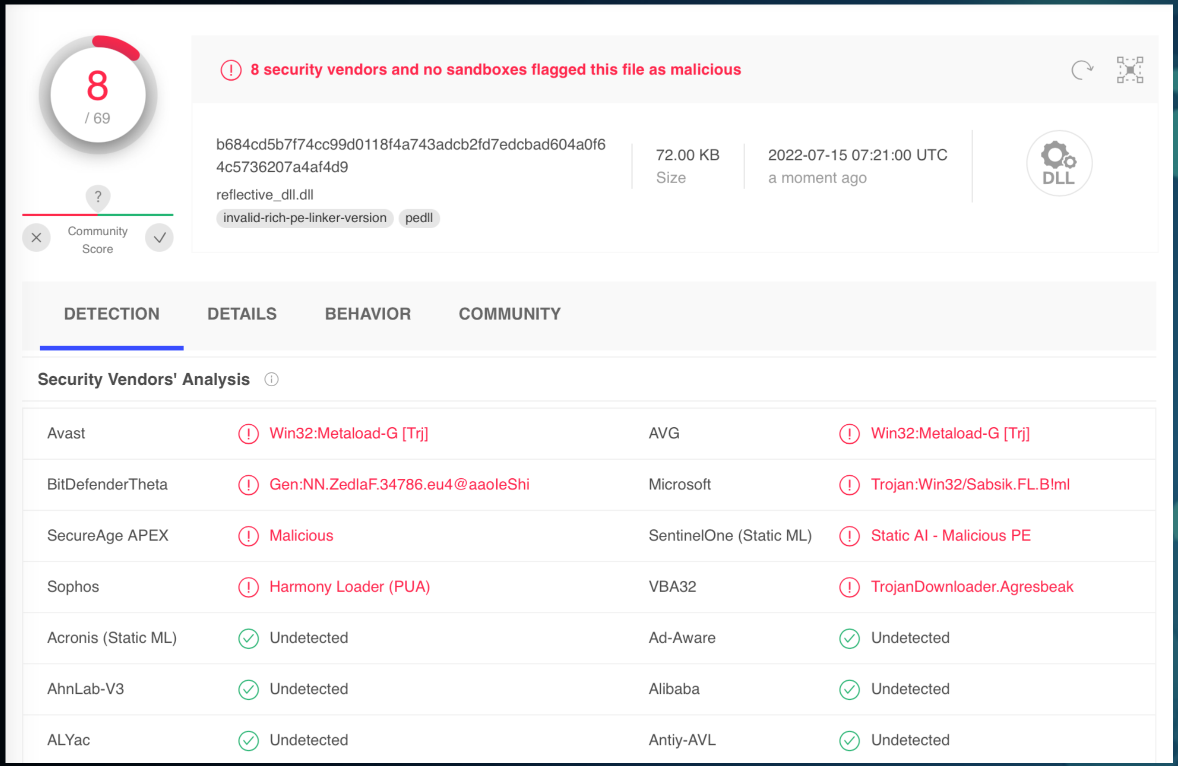
```python
# VirtualAlloc( NULL, ntHeader->OptionalHeader.SizeOfImage, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE );
if iscall and len(argv) >= 4 and argv[1] in callback.gc['sizeOfImgList']:
    callback.gc['newImageAt'] = ret

if set(argValues) & set(callback.gc['impAddrDrList']):
    callback.gc['parseIat'] = True

if set(argValues) & set(callback.gc['entryAddrList']):
    callback.gc['parseEntry'] = True
    callback.gc['entryRva'] = ( set(argValues) & set(callback.gc['entryAddrList']) ).pop()

# call the address = (new memory address) + (optionalHeader.AddressOfEntry)
if op.mnem == 'call' and callback.gc['entryRva'] + callback.gc['newImageAt'] in argValues:
    callback.gc['jmpNewImageEntry'] = True
```

```python
if callback.gc['newImageAt'] != 0xffffffff and 'parseIat' in callback.gc and 'jmpNewImageEntry' in callback.gc:
    print(f"[v] found Reflective PE Loader at {emu.funcva:x}")
    callback.gc['detect'] = True
```

txOne networks

ReflectiveLoader (Cont.)

```
exploit@exploit-lab: python3 ./TCSA/tcsa.py Sample/
b684cd5b7f74cc99d0118f4a743adcb2fd7edcbad604a0f64c5736207a4af4d9

TXOne Code Semantics Analyzer (TCSA) v1.

[OK] Rule ReflectLoader Attached.
[v] found NtHdr parsing on 100033e6 - cmp dword [ecx],0x00004550
[v] found NtHdr parsing on 10001bfd - cmp dword [eax + 268435456],0x00004550
[v] found NtHdr parsing on 100010a2 - cmp dword [ecx + esi],0x00004550
[v] found Reflective PE Loader at 10001060
[v] found NtHdr parsing on 1000b305 - cmp dword [ecx],0x00004550
  --- total used 15.06 sec ---
```



txOne networks

T1055.012 Process Hollowing

- Process Hollowing Definition from MITRE
  - Process hollowing is commonly performed by creating a process in a suspended state then unmapping/hollowing its memory, which can then be replaced with malicious code.
  - A victim process can be created with native Windows API calls such as **CreateProcess**, which includes a flag to suspend the processes primary thread. At this point the process can be unmapped using APIs calls such as **ZwUnmapViewOfSection** or **NtUnmapViewOfSection** before being written to, realigned to the injected code, and resumed via **VirtualAllocEx**, **WriteProcessMemory**, **SetThreadContext**, then **ResumeThread** respectively.
- How we collect Process Hollowing samples?
  - APT group samples from MITRE
  - APT group sample variant

https://attack.mitre.org/techniques/T1055/012/

T1055.012 Process Hollowing (Cont.)

```python
def callback(emu, starteip, op, iscall, callname, argv, argv_snapshot, ret):
    global chakraCore, ptrPInfo, ptrPeb, guessDosHdrQueue, useNewProc, useVAlloc, useWriteMem, hijackImgBase, copyHeadersToRemote

    arglist = op.getOperands()
    if not iscall and len(arglist) > 1 and arglist[1].isDeref(): # mov eax, [ebx + 0x3C] << IMAGE_DOS_HEADER.e_lfranew
        dataRef_withImmNum = getattr(arglist[1], 'disp', 0)
        if dataRef_withImmNum == 0x3C:
            guessDosHdrAddr = emu.getOperAddr(op, 1) - 0x3C
            if not guessDosHdrAddr in guessDosHdrQueue:
                print(f'[*] {hex(starteip)} - guess PE file start at (IMAGE_DOS_HEADER*) {guessDosHdrAddr}')
                guessDosHdrQueue.append(guessDosHdrAddr)
```

```python
CREATE_SUSPENDED = 0x04
if 'CreateProcess' in callname and argv[5] == CREATE_SUSPENDED:
    useNewProc = True
    ptrPInfo = argv[9]
    print(f'[*] {callname}{tuple(argv)}')
    print(f'[v] detect New Suspended Proceess ProcessInfo struct (ProcInfo) @ {(ptrPInfo)}')
```

```python
elif 'VirtualAllocEx' in callname:
    argName, argVal = argv_snapshot[0]
    hProcsRef = chakraCore.currSimulate.getAny_refOfData(argName, argVal)     # get reference of imgBase value from.

    argName, argVal = argv_snapshot[1]
    imgbasRef = chakraCore.currSimulate.getAny_refOfData(argName, argVal)     # get reference of imgBase value from.

    if hProcsRef == (ptrPInfo + 0):                                          # try to valloc on the new process?
        print(f'[*] VirtualAlloc use handle({argv[0]}) from ProcInfo.hProcess @ {hProcsRef}  ...return {ret}')
        print(f'[v] detect imagebase value from memory {imgbasRef}')
        useVAlloc = True
```

txOne networks

T1055.012 Process Hollowing (Cont.)

```python
elif 'GetThreadContext' in callname:
    argName, argVal = argv_snapshot[0]
    hProcsRef = chakraCore.currSimulate.getAny_refOfData(argName, argVal)
    if hProcsRef == (ptrPInfo + 4):                     # try to get suspended thread context of the new process?
        ptrPeb = argv[1] + 0xA4                         # offset CONTEXT.ebx = 0xA4
```

```python
elif 'WriteProcessMemory' in callname:
    argName, argVal = argv_snapshot[0]
    hProcsRef = chakraCore.currSimulate.getAny_refOfData(argName, argVal)

    # try to write memory of the new process?
    if hProcsRef == (ptrPInfo + 0):
        useWriteMem = True

        # where're you writing at? is that CONTEXT.ebx (PEB) + 8?
        argName, argVal = argv_snapshot[1]
        if pebAddrRef := chakraCore.currSimulate.getAny_refOfData(argName, argVal):
            if _ := ptrPeb and pebAddrRef == ptrPeb + 8:
                hijackImgBase = True
                print(f'[v] detect write remote PEB.imagebase to hijack main module.')

        # trying to copy DOS+NT+Section headers 3 blocks to remote?
        if argv[2] in guessDosHdrQueue:
            argName, argVal = argv_snapshot[3]
            dataSizeRef = chakraCore.currSimulate.getAny_refOfData( argName, argVal)
            guessNtHdrAddr = argv[2] + emu.readMemoryPtr( argv[2] + 0x3c )
            if guessNtHdrAddr + 0x18 + 0x3C == dataSizeRef:  # is that size from NtHdr.OptionalHeader(+18h).SizeOfHeaders(+3Ch)
                print('[v] detect copy PE headers to remote, include DOS+NT+Sections.')
                copyHeadersToRemote = True
```

```python
def cleanup( In_chakraCore, In_capaMatchRet ):
    if useNewProc and useVAlloc and useWriteMem and hijackImgBase and copyHeadersToRemote:
        print(' !!! Assert That should be Hollowing Tricks !!! ')

    print(f'\n === [Capability-Detection] === ')
    print(f' Create Suspended Process                     : {useNewProc}')
    print(f' Malloc Memory at NtHdr.OptionalHeader.Imgbase : {useNewProc}')
    print(f' Hijack ImageBase of Main PE Module           : {hijackImgBase}')
    print(f' Copy PE Headers (DOS, NT, Sections) to Remote : {copyHeadersToRemote}')
```

txOne networks

T1055.012 Process Hollowing (Cont.)

```
exploit@exploit-lab: python3 ./TCSA/tcsa.py Sample/1c64966bdcbc55db0256a1aa3fc99062ba1837849b1cc5aa59ce0e31bf279e09

TXOne Code Semantics Analyzer (TCSA) v1.
[OK] Rule Attached - Process Hollowing.
[v] 0x40b367 - guess PE file start at PIMAGE_DOS_HEADER( 0xbfb07f84 )
[v] 0x40235b - guess PE file start at PIMAGE_DOS_HEADER( 0x4156100f )
[v] 0x402b40 - guess PE file start at PIMAGE_DOS_HEADER( 0xbfb07f30 )
[v] 0x401652 - kernel32.CreateProcessW(0x0, 0x4157100f, 0x0, 0x0, 0x0, 0x4, 0x0, 0x0, 0xbfb07c84, 0xbfb07c74)
[v] detect New Suspended Proceess ProcessInfo struct (ProcInfo) @ 0xbfb07c74
[v] VirtualAlloc use handle(0) from ProcInfo.hProcess @ 0xbfb07c74  ...return 0x4159f00f
[v] detect imagebase value from memory 0xa2b891a4
[v] detect copy PE headers to remote, include DOS+NT+Sections.
[v] detect write remote PEB.imagebase to hijack main module.
 !!! Assert That should be Hollowing Tricks !!!

 === [Capability-Detection] ===
 Create Suspended Process                     : True
 Malloc Memory at NtHdr.OptionalHeader.Imgbase : True
 Hijack ImageBase of Main PE Module           : True
 Copy PE Headers (DOS, NT, Sections) to Remote : True
  --- total used 11.73 sec ---
```

```
CreateProcessW(0, lpCommandLine, 0, 0, 0, CREATE_SUSPENDED, 0, 0, &StartupInfo, &ProcessInformation);
sprintf(byte_414060, "%S%S", L"ZwUnmapView", L"OfSection");
v2 = *(a2 + 60);
v3 = GetProcAddress(dword_414020, byte_414060);
v4 = a2 + v2;
v5 = *(v4 + 52);
dword_414048 = v3;
(v3)(ProcessInformation.hProcess, v5, v6, v7, v8);
sprintf(byte_414060, "Vir%s%scEx", "tual", "Allo");
v9 = GetProcAddress(hModule, byte_414060);
v10 = *(v4 + 80);
dword_414038 = v9;
v11 = *(v4 + 52);
v12 = v9();
v22 = v12;
if ( v12 )
{
  sprintf(byte_414060, "Wr%sProcess%sory", "ite", "Mem");
  v13 = GetProcAddress(hModule, byte_414060);
  v14 = *(v4 + 84);
  dword_41404C = v13;
  v15 = v12;
  v16 = 0;
  (v13)(ProcessInformation.hProcess, v15, a2, v14, 0);
  if ( *(v4 + 6) )
  {
    do
    {
      v17 = 5 * v16++;
      v18 = (a2 + *(a2 + 60) + 248 + 8 * v17);
      dword_41404C(ProcessInformation.hProcess, v18[3] + v22, a2 + v18[5], v18[4], 0);
    }
    while ( *(v4 + 6) > v16 );
  }
  v19 = ProcessInformation.hThread;
  v25.ContextFlags = 65543;
  *dword_41403C = GetProcAddress(hModule, "GetThreadContext");
  dword_41403C(v19, &v25);
  dword_41404C(ProcessInformation.hProcess, v25.Ebx + 8, v4 + 52, 4, 0);
  v25.Eax = *(v4 + 40) + v22;
  dword_414040 = GetProcAddress(hModule, "SetThreadContext");
  (dword_414040)(ProcessInformation.hThread, &v25);
```

T1055.012 Process Hollowing

- Process Hollowing Definition from MITRE
  - Process hollowing is commonly performed by creating a process in a suspended state then unmapping/hollowing its memory, which can then be replaced with malicious code
  - A victim process can be created with native Windows API calls such as **CreateProcess**, which includes a flag to suspend the processes primary thread. At this point the process can be unmapped using APIs calls such as **ZwUnmapViewOfSection** or **NtUnmapViewOfSection** before being written to, realigned to the injected code, and resumed via **VirtualAllocEx**, **WriteProcessMemory**, **SetThreadContext**, then **ResumeThread** respectively
- How we collect Process Hollowing samples?
  - APT group samples from MITRE
  - APT group sample variant
- The TCSA Rule can detect obfuscated samples (striped symbols)

https://attack.mitre.org/techniques/T1055/012/

T1055.012 Process Hollowing (Cont.)

```
//process in suspended state, for the new image.
if (CreateProcessA(szBenign, 0, 0, 0, 0, CREATE_SUSPENDED, NULL, NULL, &SI, &PI)) {

    // Allocate memory for the context.
    CTX = LPCONTEXT(VirtualAlloc(NULL, sizeof(CTX), MEM_COMMIT, PAGE_READWRITE));
    CTX->ContextFlags = CONTEXT_FULL; // Context is allocated

    if ( GetThreadContext(PI.hThread, LPCONTEXT(CTX)) ){

        pImageBase = VirtualAllocEx(PI.hProcess, LPVOID(NtHeader->OptionalHeader.ImageBase),
            NtHeader->OptionalHeader.SizeOfImage, 0x3000, PAGE_EXECUTE_READWRITE);

        // Write the image to the process
        WriteProcessMemory(PI.hProcess, pImageBase, Image, NtHeader->OptionalHeader.SizeOfHeaders, NULL);
        for (count = 0; count < NtHeader->FileHeader.NumberOfSections; count++)
            WriteProcessMemory(
                PI.hProcess, LPVOID(DWORD(pImageBase) + SectionHeader->VirtualAddress),
                LPVOID(DWORD(Image) + SectionHeader->PointerToRawData), SectionHeader->SizeOfRawData, 0);

        // Switch PEB.ImageBase to our malicious PE Image Addr
        WriteProcessMemory(PI.hProcess, LPVOID(CTX->Ebx + 8), LPVOID(&NtHeader->OptionalHeader.ImageBase), 4, 0);

        // Move address of entry point to the eax register
        CTX->Eax = DWORD(pImageBase) + NtHeader->OptionalHeader.AddressOfEntryPoint;
        SetThreadContext(PI.hThread, LPCONTEXT(CTX)); // Set the context
        ResumeThread(PI.hThread); //´Start the process/call main()
```

- Create a suspended victim process by CreateProcess
- Mount malicious modules in its memory
- Get the register EBX value by GetThreadContext
  - The register EBX value will point to the PEB structure address of that process.
- Modify the ImageBase on the PEB structure by WriteProcessMemory
  - Switching the main executed PE module to the malicious module
- Modify the EAX register so the execution entry jump to the malware entry

Striped Process Hollowing

```python
def callback(emu, starteip, op, iscall, callname, argv, argv_snapshot, ret):
    arglist = op.getOperands()

    if iscall and len(argv) >= 10 and argv[2] == argv[3] == argv[4] == 0:
        useNewProc = True
        ptrPInfo = argv_snapshot[9]
        callback.list_spawnProc.append( (emu.funcva, starteip) ) # ( funcva, createProcess_callAt )
        print(f'[*] spawn process? {starteip:x} @ sub_{emu.funcva:x} - {callname}{tuple(argv)}')

callback.list_spawnProc = []
```

```python
def cleanup( In_chakraCore, In_capaMatchRet ):

    # check the each potential CreateProcess usage.
    for funcAddr, newProc_callAt in callback.list_spawnProc:

        # hollowing detection state-machine
        def stateMachine_hollowing(emu, eip, op, iscall, argv):...

        stateMachine_hollowing.guess_pebImageBaseAt = set()
        stateMachine_hollowing.useCtxFlag_CTXFULL = False

        # verify the behavior of the each caller.
        for callerFunc, argSnapshot in chakraCore.lookup_Caller(funcAddr):
            chakraCore.tinySimulateSingleFunction(callerFunc, stateMachine_hollowing)

        # verify the function which contains the potential CreateProcess
        # means that function might not have any parent function, it's a entry function?
        chakraCore.tinySimulateSingleFunction(funcAddr, stateMachine_hollowing)
```

Striped Process Hollowing (Cont.)

```python
# hollowing detection state-machine
def stateMachine_hollowing(emu, eip, op, iscall, argv):
    modifyState = False
    arglist = set( [ emu.getOperValue(op, _) for _ in range(len(op.opers)) ] )

    if eip == newProc_callAt and len(argv) >= 9 and argv[5] == 0x04 and checkNum_isData(emu, argv[9]):
        guess_procInfoAt = argv[9] # lpProcessInformation
        emu.allocateMemory(256, suggestaddr=guess_procInfoAt) # ensure the struct is allocated in memory
        emu.writeMemoryPtr( guess_procInfoAt + 0, 0xDEADDEAD ) # set hProcess to 0xDEADDEAD
        emu.writeMemoryPtr( guess_procInfoAt + 4, 0xBEEFBEEF ) # set hThread to 0xBEEFBEEF
        modifyState = True

    # [CASE] CONTEXT.ContextFlags = CONTEXT_FULL
    set_CONTEXTFLAGS = set([ 0x10007, 0x1003F ])
    stateMachine_hollowing.useCtxFlag_CTXFULL |= {} != set_CONTEXTFLAGS & arglist

    # [CASE] GetThreadContext( 0xBEEFBEEF, &CONTEXT )
    if iscall and len(argv) >= 2 \
       and argv[0] == 0xBEEFBEEF and checkNum_isData(emu, argv[1]):

        ebxVal = emu.readMemoryPtr(argv[1] + 0xA4) # offsetof(CONTEXT, Ebx) = A4h
        stateMachine_hollowing.guess_pebImageBaseAt.add(ebxVal + 8)

    if arglist & stateMachine_hollowing.guess_pebImageBaseAt:
        print(f"[v] found access PEB.ImageBase at {eip:x} - {op}")

    # [TRUE]:  keep the modified execution-state if we're doing some kinda necessary patchs.
    # [FALSE]: state-machine will forgot all the memory patchs
    #          when running out of the current function scope. (back to the parent function)
    return modifyState
```

txOne networks

Striped Process Hollowing (Cont.)

- Experiment
- How we collect Hollowing samples?
  - Time interval: 2022.1.1~Now
  - Filter process
    - Find in VirusTotal, behaviour_injected_processes
    - More than 10 antivirus vendors, and it is Windows executable
    - Using Classic Process Hollowing Definition (based on MITRE) and not packed.
  - Results
    - 141 / 233 -> 60.51% of injection samples from VirusTotal should be hollowing.
      -> 39.49% Based on manual analysis, verified not hollowing samples.
      Cheat Engine, x64dbg, Chrome Installer ...

txOne networks

Real World Ransomware Detection (Cont.)

- Basically, ransomware does the following capability
  - Find unfamiliar files (such as FindFirstFile)
  - Read/Write behavior in the same file (such as CreateFile -> ReadFile -> SetFilePointer ->WriteFile)
  - Identify common encrypt function or algorithm (WinCrypt*, AES, ChaCha, RC4…)
- What are our criteria of detection?
  - 3 features (file enumeration, file operations, encryption) detected or
  - One of the chain
    - File enumeration → Encryption
    - File enumeration & File operations → Encryption

txOne networks

Real World Ransomware Detection (Cont.)

- File Enumerate

```
bool ransomMain(void)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-

  strcpy(aesKey, "3igcZhRdWq96m3GUmTAiv9");
  hFind = FindFirstFileA("*.*", &FindFileData);
  while ( 1 )
  {
    result = FindNextFileA(hFind, &FindFileData);
    if ( !result )
      break;
    if ( FindFileData.cFileName[0] != '.' )
    {
      strcat(pathToFile, FindFileData.cFileName);
      encryptFile(pathToFile, aesKey, 0x17u);
      printf("[v] encrypt file - %s\n", pathToFile);
    }
  }
  return result;
}
```

WannaCry Ransomware sample via IDA Pro

```
def callback(emu, starteip, op, iscall, callname, argv, argv_snapshot, ret):

    if emu.funcva not in guessList_findDataStruct:
        guessList_findDataStruct[emu.funcva], guessList_fileData_cFileName[emu.funcva] = [], []

    if iscall:
        arg1, arg2, arg3 = argv[0], argv[1], argv[2]

        if "FindFirstFileA" == callname or "FindFirstFileW" == callname \
        or ( len(argv) >= 2 and isPointer(emu, arg1) and (isPointer(emu, arg2) or arg2 == 0) ):
            guessList_findDataStruct[emu.funcva].append( ret )

        if "FindNextFileA" == callname or "FindNextFileW" == callname \
        or ( len(argv) >= 2 and arg1 in guessList_findDataStruct[emu.funcva] ) and isPointer(emu, arg2):
            guessList_fileData_cFileName[emu.funcva].append(arg2 + 0x2C) # FindFileData.cFileName (+2Ch)


    if len(op.opers) > 1:
        if emu.getOperAddr(op, 1)  in guessList_fileData_cFileName[emu.funcva] \
        or emu.getOperValue(op, 1) in guessList_fileData_cFileName[emu.funcva] :
            print(f'[+] fva: {hex(emu.funcva)}, Taint FileData.cFileName: {hex(starteip)}')
```

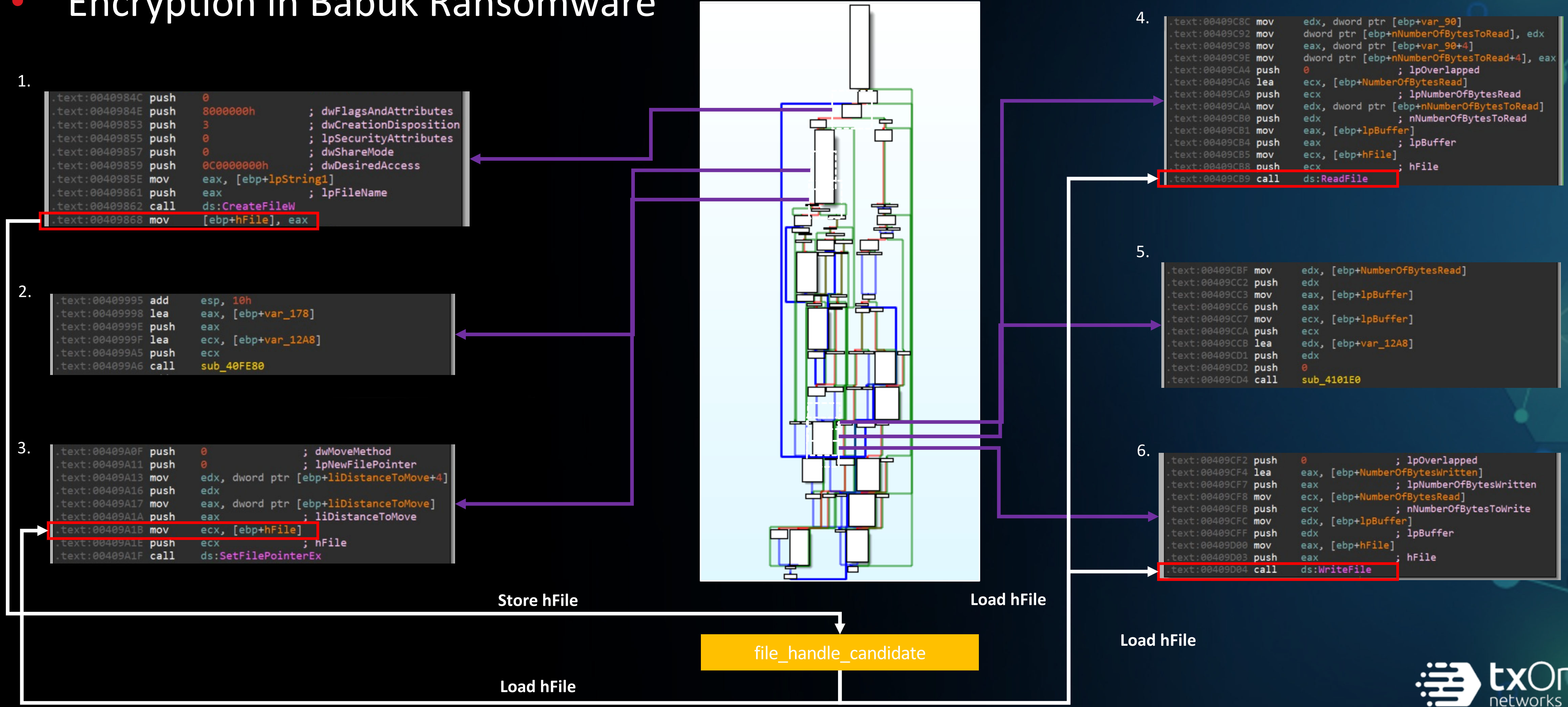
Real World Ransomware Detection (Cont.)

- Taint file handle generated from CreateFile*
  - Monitor file I/O API usage

```
def callback(emu, starteip, op, iscall, callname, argv, argv_snapshot, ret):


    if ("CreateFileA" in callname) or ("CreateFileW" in callname) or \
    ((len(argv) >= 7) and \
    not isPointer(emu, argv[1]) and (argv[1] & 0xFFFFFFFF & (GENERIC_READ | GENERIC_WRITE | GENERIC_ALL)) and \
    not isPointer(emu, argv[2]) and (argv[2] == 0 or argv[2] & 0xFFFFFFFF & (FILE_SHARE_LOCK | FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE)) and \
    not isPointer(emu, argv[4]) and (argv[4] & 0xFFFFFFFF in (CREATE_ALWAYS, OPEN_EXISTING, CREATE_NEW, OPEN_ALWAYS)) and \
    not isPointer(emu, argv[5])):

        record_handle(file_handle_list, emu.funcva, ret, starteip)
        record_handle(file_handle_candidate, emu.funcva, ret, starteip)


    if ("SetFilePointer" in callname) or \
    ((len(argv) >= 4) and argv[3] == 0): # FILE_BEGIN
        record_handle(file_handle_candidate, emu.funcva, argv[0], starteip)


    if ("ReadFile" in callname) or ("WriteFile" in callname) or \
    ((len(argv) >= 5) and isPointer(emu, argv[1])):
        record_handle(file_handle_candidate, emu.funcva, argv[0], starteip)
```

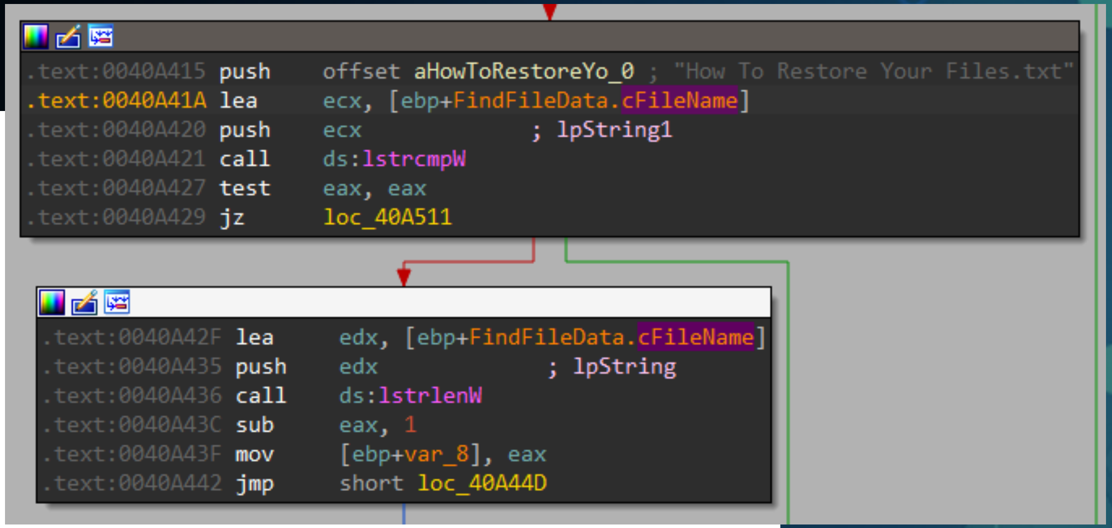
Real World Ransomware Detection (Cont.)

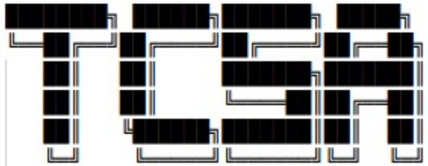
- Encryption in Babuk Ransomware



Real World Ransomware Detection (Cont.)

- Babuk Ransomware – File Enumeration





```
TXOne Code Semantics Analyzer (TCSA) v1.
[<module 'Plugins' from '/home/hank/TCSA/Plugins/rule_ransomware.py'>]
[OK] Rule Ransomware Attached.
[+] fva: 0x40a5e0, Taint FileData.cFileName: 0x40a6ef
[+] fva: 0x40a5e0, Taint FileData.cFileName: 0x40a6bb
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a41a
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a42f
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a3bb
[+] fva: 0x404a80, create new key via CryptAcquireContext
[+] fva: 0x409740, generate random numbers via WinAPI
[+] fva: 0x40fe80, encrypt data using HC-128 wrapper
[+] fva: 0x409740, CreateFile addr: ['0x409d63'], Taint Handle: ['0x409894', '0x409d67']
[+] fva: 0x409740, CreateFile addr: ['0x409c7a', '0x409c8c', '0x409caa', '0x409c63', '0x409b54', '0x409a49'], Taint Handle: ['0x409c67', '0x409b58', '0x409a4d']
[+] fva: 0x40a2d0, CreateFile addr: ['0x40a323', '0x40a349', '0x40a353'], Taint Handle: ['0x40a323', '0x40a34d', '0x40a357']
========== function topology ==========
[file->encrypt] depth: 0, chain: ['0x409740']
[file->encrypt] depth: 1, chain: ['0x409740', '0x40fe80']
[file->encrypt] depth: 1, chain: ['0x40a2d0', '0x409740']
[file->encrypt] depth: 2, chain: ['0x40a2d0', '0x409740', '0x40fe80']
[enum->encrypt] depth: 1, chain: ['0x40a5e0', '0x409740']
[enum->encrypt] depth: 2, chain: ['0x40a5e0', '0x409740', '0x40fe80']
[enum->encrypt] depth: 1, chain: ['0x40a2d0', '0x409740']
[enum->encrypt] depth: 2, chain: ['0x40a2d0', '0x409740', '0x40fe80']
 --- total used 13.150455474853516 sec ---
```

txOne networks

Real World Ransomware Detection (Cont.)

- Babuk Ransomware – File Operation

```
TXOne Code Semantics Analyzer (TCSA) v1.
[<module 'Plugins' from '/home/hank/TCSA/Plugins/rule_ransomware.py'>]
[OK] Rule Ransomware Attached.
[+] fva: 0x40a5e0, Taint FileData.cFileName: 0x40a6ef
[+] fva: 0x40a5e0, Taint FileData.cFileName: 0x40a6bb
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a41a
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a42f
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a3bb
[+] fva: 0x404a80, create new key via CryptAcquireContext
[+] fva: 0x409740, generate random numbers via WinAPI
[+] fva: 0x40fe80, encrypt data using HC-128 wrapper
[+] fva: 0x409740, CreateFile addr: ['0x409d63'], Taint Handle: ['0x409894', '0x409d67']
[+] fva: 0x409740, CreateFile addr: ['0x409c7a', '0x409c8c', '0x409caa', '0x409c63', '0x409b54', '0x409a49'], Taint Handle: ['0x409c67', '0x409b58', '0x409a4d']
[+] fva: 0x40a2d0, CreateFile addr: ['0x40a323', '0x40a349', '0x40a353'], Taint Handle: ['0x40a323', '0x40a34d', '0x40a357']
========== function topology ==========
[file->encrypt] depth: 0, chain: ['0x409740']
[file->encrypt] depth: 1, chain: ['0x409740', '0x40fe80']
[file->encrypt] depth: 1, chain: ['0x40a2d0', '0x409740']
[file->encrypt] depth: 2, chain: ['0x40a2d0', '0x409740', '0x40fe80']
[enum->encrypt] depth: 1, chain: ['0x40a5e0', '0x409740']
[enum->encrypt] depth: 2, chain: ['0x40a5e0', '0x409740', '0x40fe80']
[enum->encrypt] depth: 1, chain: ['0x40a2d0', '0x409740']
[enum->encrypt] depth: 2, chain: ['0x40a2d0', '0x409740', '0x40fe80']
 --- total used 13.150455474853516 sec ---
```

```
.text:0040A309 push    edx               ; lpString1
.text:0040A30A call    ds:lstrcatW
.text:0040A310 push    0                 ; hTemplateFile
.text:0040A312 push    0                 ; dwFlagsAndAttributes
.text:0040A314 push    1                 ; dwCreationDisposition
.text:0040A316 push    0                 ; lpSecurityAttributes
.text:0040A318 push    1                 ; dwShareMode
.text:0040A31A push    40000000h         ; dwDesiredAccess
.text:0040A31F mov     eax, [ebp+lpString1]
.text:0040A322 push    eax               ; lpFileName
.text:0040A323 call    ds:CreateFileW
.text:0040A329 mov     [ebp+hFile], eax
```

txOne networks

Real World Ransomware Detection (Cont.)

- Babuk Ransomware – File Encryption

```
TXOne Code Semantics Analyzer (TCSA) v1.
[<module 'Plugins' from '/home/hank/TCSA/Plugins/rule_ransomware.py'>]
[OK] Rule Ransomware Attached.
[+] fva: 0x40a5e0, Taint FileData.cFileName: 0x40a6ef
[+] fva: 0x40a5e0, Taint FileData.cFileName: 0x40a6bb
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a41a
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a42f
[+] fva: 0x40a2d0, Taint FileData.cFileName: 0x40a3bb
[+] fva: 0x404a80, create new key via CryptAcquireContext
[+] fva: 0x409740, generate random numbers via WinAPI
[+] fva: 0x40fe80, encrypt data using HC-128 wrapper
[+] fva: 0x409740, CreateFile addr: ['0x409d63'], Taint Handle: ['0x409894', '0x409d67']
[+] fva: 0x409740, CreateFile addr: ['0x409c7a', '0x409c8c', '0x409caa', '0x409c63', '0x409b54', '0x409a49'], Taint Handle: ['0x409c67', '0x409b58', '0x409a4d']
[+] fva: 0x40a2d0, CreateFile addr: ['0x40a323', '0x40a349', '0x40a353'], Taint Handle: ['0x40a323', '0x40a34d', '0x40a357']
========== function topology ==========
[file->encrypt] depth: 0, chain: ['0x409740']
[file->encrypt] depth: 1, chain: ['0x409740', '0x40fe80']
[file->encrypt] depth: 1, chain: ['0x40a2d0', '0x409740']
[file->encrypt] depth: 2, chain: ['0x40a2d0', '0x409740', '0x40fe80']
[enum->encrypt] depth: 1, chain: ['0x40a5e0', '0x409740']
[enum->encrypt] depth: 2, chain: ['0x40a5e0', '0x409740', '0x40fe80']
[enum->encrypt] depth: 1, chain: ['0x40a2d0', '0x409740']
[enum->encrypt] depth: 2, chain: ['0x40a2d0', '0x409740', '0x40fe80']
 --- total used 13.150455474853516 sec ---
```

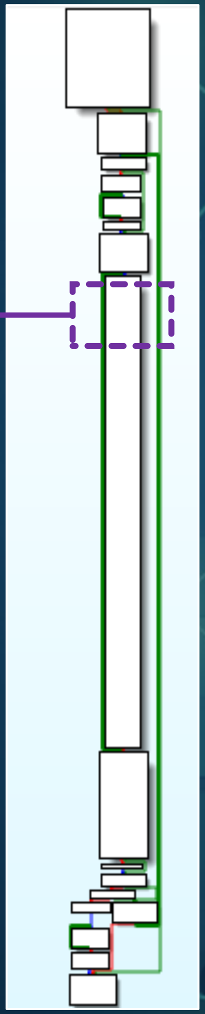
```
.text:0040FE80 ; Attributes: bp-based frame
.text:0040FE80
.text:0040FE80 sub_40FE80 proc near
.text:0040FE80
.text:0040FE80 var_4= dword ptr -4
.text:0040FE80 arg_0= dword ptr  8
.text:0040FE80 arg_4= dword ptr  0Ch
.text:0040FE80
.text:0040FE80 push    ebp
.text:0040FE81 mov     ebp, esp
.text:0040FE83 push    ecx
.text:0040FE84 push    esi
.text:0040FE85 push    edi
.text:0040FE86 mov     [ebp+var_4], 0
.text:0040FE8D jmp     short loc_40FE98

.text:0040FE98
.text:0040FE98 loc_40FE98:
.text:0040FE98 mov     ecx, [ebp+arg_0]
.text:0040FE9B mov     edx, [ecx+10C8h]
.text:0040FEA1 shr     edx, 5
.text:0040FEA4 cmp     [ebp+var_4], edx
.text:0040FEA7 jnb     short loc_40FEC1
```

txOne networks

Real World Ransomware Detection (Cont.)

- How we improve the detection rate?
  - Darkside
    - Customized Salsa20 matrix and encryption
    - 4 rounds of linear shifting

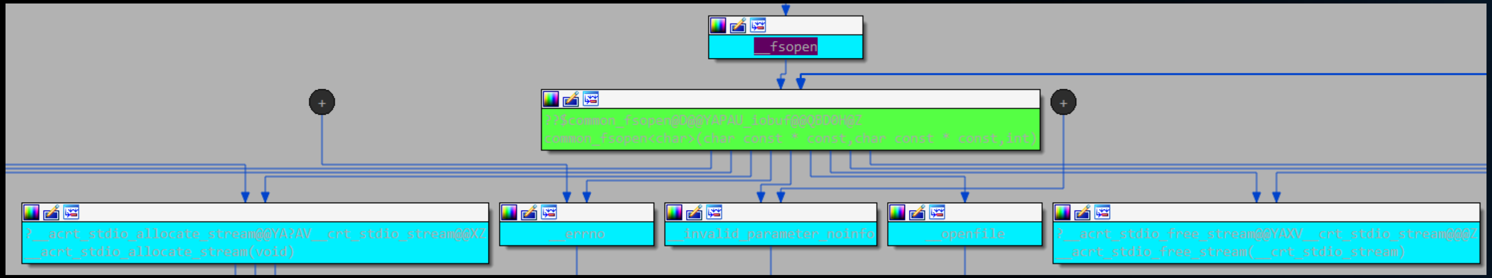
```
.text:00402187 mov     eax, [edi]
.text:00402189 mov     ebx, [edi+10h]
.text:0040218C mov     ecx, [edi+20h]
.text:0040218F mov     edx, [edi+30h]
.text:00402192 mov     esi, eax
.text:00402194 add     esi, edx
.text:00402196 rol     esi, 7
.text:00402199 xor     ebx, esi
.text:0040219B mov     esi, ebx
.text:0040219D add     esi, eax
.text:0040219F rol     esi, 9
.text:004021A2 xor     ecx, esi
.text:004021A4 mov     esi, ecx
.text:004021A6 add     esi, ebx
.text:004021A8 rol     esi, 0Dh
.text:004021AB xor     edx, esi
.text:004021AD mov     esi, edx
.text:004021AF add     esi, ecx
.text:004021B1 rol     esi, 12h
.text:004021B4 xor     eax, esi
.text:004021B6 mov     [edi], eax
.text:004021B8 mov     [edi+10h], ebx
.text:004021BB mov     [edi+20h], ecx
.text:004021BE mov     [edi+30h], edx
```



txOne networks

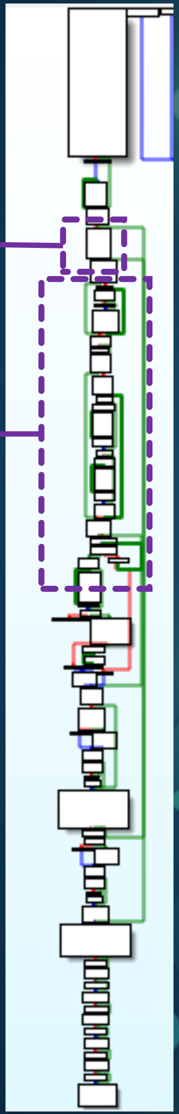
Real World Ransomware Detection (Cont.)

- How we improve the detection rate?
  - 7ev3n
    - R5A Encryption
    - fsopen() from msvcrt



Check if the first byte is 'M'

Extend stream cipher key from filename and encrypt the file content



txOne networks

Real World Ransomware Detection (Cont.)

- Experiment
- How we collect Ransomware samples?
  - Time interval: 2021.06-2022.06
  - Filter process
    - Found in VirusTotal, more than 3 antivirus vendors identify ransomware, and it is Windows executable
    - Automated dynamic analysis (commercial sandbox)
    - Final check samples
    - Get ransomware sample dataset
  - Results
    - 1153 / 1206 (95.60%) !!!

txOne networks

Real World Ransomware Detection (Cont.)

Purge	Seven	Phobos	Lockbit	Agent	Explus	Taleb	Hive
Rents	Medusalocker	Cryptolocker	Makop	Redeemer	Sodinokibi	Garrantycrypt	Tovicrypt
Conti	Crysis	Filecoder	Crypren	Hydracrypt	Avoslocker	Sevencrypt	Crypmod
Sorikrypt	Higuniel	Paradise	Cryptor	Wixawm	Zcrypt	Sodinokib	Xorist
Nemty	Fakeglobe	Emper	Quantumlocker	Blackmatter	Revil	Bastacrypt	Ranzylocker
Avaddon	Netfilm	Wana	Garrantdecrypt	Smar	Akolocker	Cryptlock	Wadhrama
Phoenix	Spora	Babuklocker	Lockergoga	Buhtrap	Ryuk	Nemisis	Netwalker
Deltalocker	Karmalocker	Genasom	Thundercrypt	Wcry	Hkitty	Swrort	Babuk

txOne networks

Real World Ransomware Detection (Cont.)

- Conti variants

  Ransom.Win32.CONTI.SM.hp
  Ransom.Win32.CONTI.SMTH.hp
  Ransom.Win32.CONTI.SMYXBBU
  Ransom.Win32.CONTI.SMYXBFD.hp
  Ransom.Win32.CONTI.YACCA
  Ransom.Win32.CONTI.YXCAAZ
  Ransom.Win32.CONTI.YXCBSZ

- LockBit variants

  Ransom.Win32.LOCKBIT.SMCET
  Ransom.Win32.LOCKBIT.SMDS
  Ransom.Win32.LOCKBIT.SMYEBGW
  Ransom.Win32.LOCKBIT.YXBHC-TH
  Ransom_LockBit.R002C0CGI21
  Ransom_Lockbit.R002C0DCO22
  Ransom_Lockbit.R002C0DHB21
  Ransom_Lockbit.R002C0DHD21

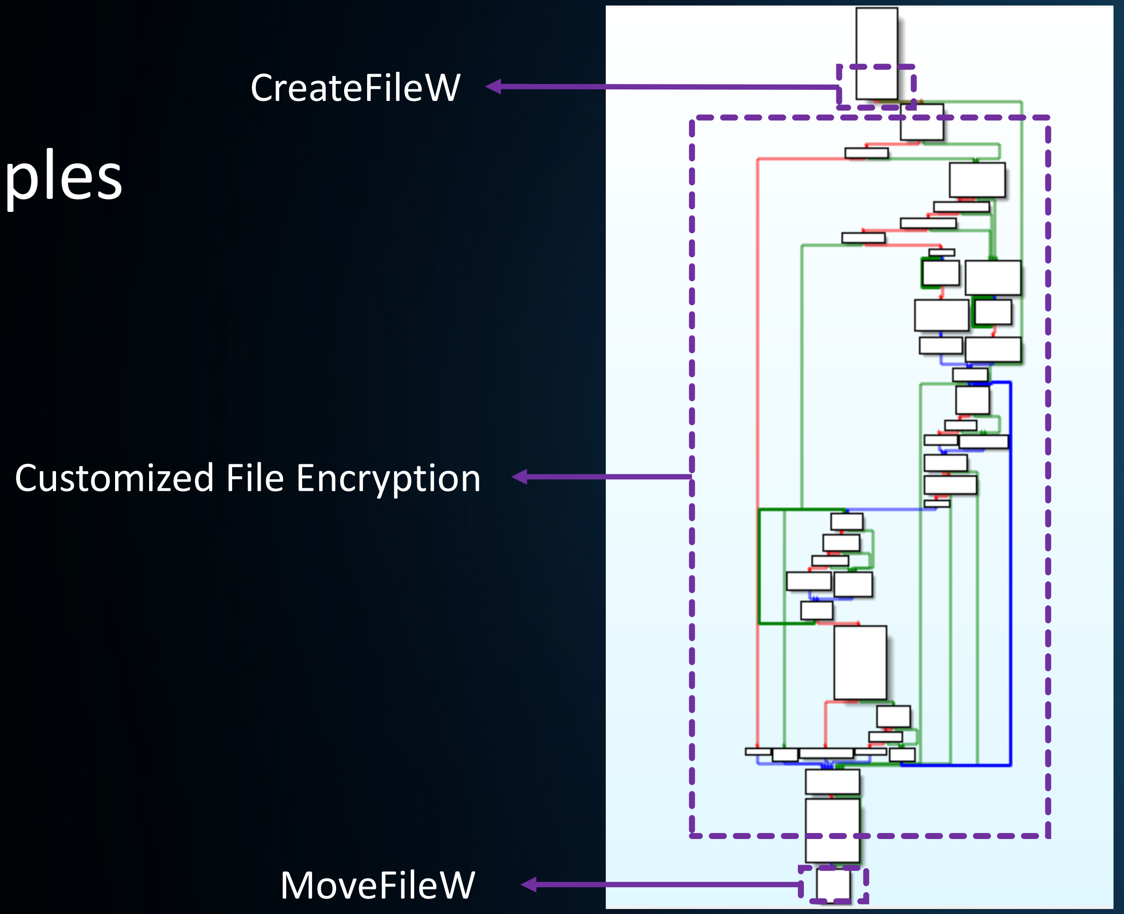
- 7ev3n variants

  Ransom_Seven.R002C0DA422
  Ransom_Seven.R002C0DA522
  Ransom_Seven.R002C0DA922
  Ransom_Seven.R002C0DAA22
  Ransom_Seven.R002C0DAF22
  Ransom_Seven.R002C0DAP22
  Ransom_Seven.R002C0DAR22
  Ransom_Seven.R002C0DAS22
  Ransom_Seven.R002C0DAT22
  Ransom_Seven.R002C0DAV22
  Ransom_Seven.R002C0DB122
  Ransom_Seven.R002C0DB222
  Ransom_Seven.R002C0DB322
  Ransom_Seven.R002C0DB822
  Ransom_Seven.R002C0DB922
  Ransom_Seven.R002C0DBA22
  Ransom_Seven.R002C0DBM22
  Ransom_Seven.R002C0DC222
  Ransom_Seven.R002C0DC922
  Ransom_Seven.R002C0DCB22
  Ransom_Seven.R002C0DCC22
  Ransom_Seven.R002C0DCE22
  Ransom_Sodin.R002C0PGM21
  Ransom_EMPER.SM

txOne networks

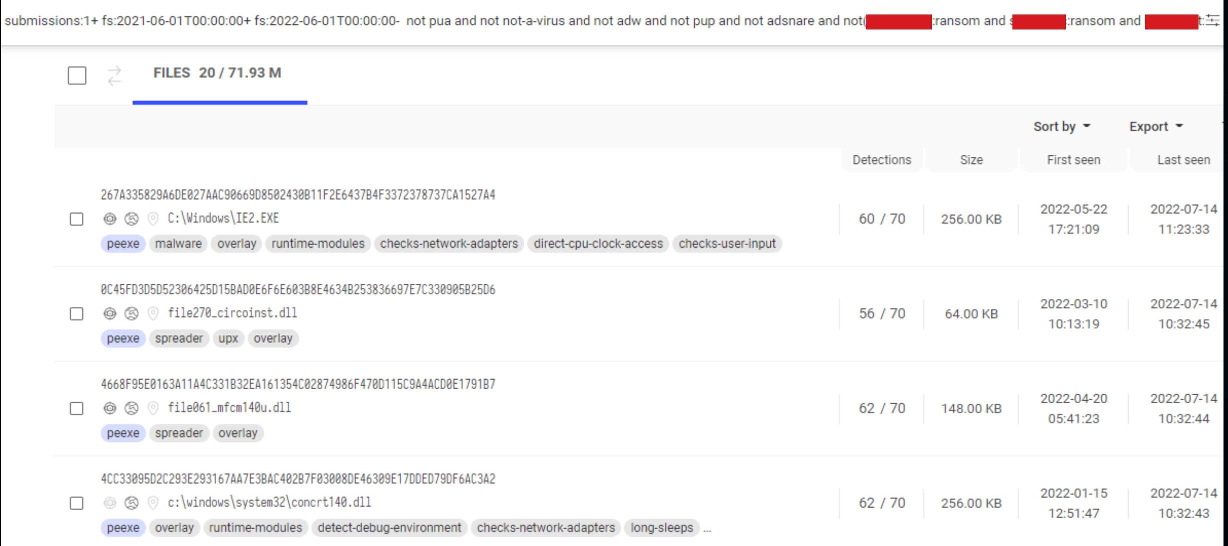
Real World Ransomware Detection (Cont.)

- For some of undetected samples
  - Prolock / PwndLocker
    - Unknown Encryption Algorithm



Real World Ransomware Detection (Cont.)

- Experiment
  - By randomly finding 200 non-ransom samples from VirusTotal (2021/06/01 - 2022/06/01)
    - False Positive: 0%

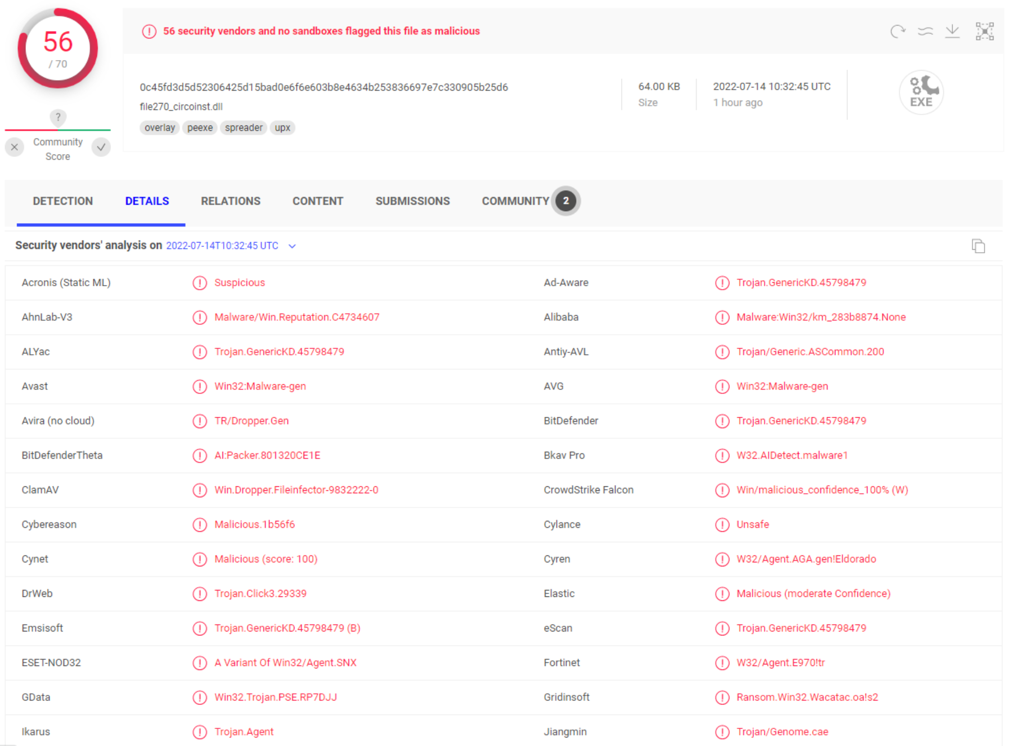


submissions:1+ fs:2021-06-01T00:00:00+ fs:2022-06-01T00:00:00- not pua and not not-a-virus and not adw and not pup and not adsnare and not [redacted] ransom and [redacted] ransom and [redacted]

FILES 20 / 71.93 M

Sort by   Export

	Detections	Size	First seen	Last seen
267A335829A6DE027AAC90669D8502430B11F2E6437B4F3372378737CA1527A4 C:\Windows\IE2.EXE peexe malware overlay runtime-modules checks-network-adapters direct-cpu-clock-access checks-user-input	60 / 70	256.00 KB	2022-05-22 17:21:09	2022-07-14 11:23:33
0C45FD3D5D52306425D15BAD0E6F6E603B8E4634B253836697E7C330905B25D6 file270_circoinst.dll peexe spreader upx overlay	56 / 70	64.00 KB	2022-03-10 10:13:19	2022-07-14 10:32:45
4668F95E0163A11A4C331B32EA161354C02874986F470D115C9A4ACD0E1791B7 file061_mfcm140u.dll peexe spreader overlay	62 / 70	148.00 KB	2022-04-20 05:41:23	2022-07-14 10:32:44
4CC33095D2C293E293167AA7E3BAC402B7F03008DE46309E17DDED79DF6AC3A2 c:\windows\system32\concrt140.dll peexe overlay runtime-modules detect-debug-environment checks-network-adapters long-sleeps ...	62 / 70	256.00 KB	2022-01-15 12:51:47	2022-07-14 10:32:43



56 / 70

56 security vendors and no sandboxes flagged this file as malicious

0c45fd3d5d52306425d15bad0e6f6e603b8e4634b253836697e7c330905b25d6
file270_circoinst.dll
overlay peexe spreader upx

64.00 KB Size

2022-07-14 10:32:45 UTC 1 hour ago

EXE

Community Score

DETECTION DETAILS RELATIONS CONTENT SUBMISSIONS COMMUNITY 2

Security vendors' analysis on 2022-07-14T10:32:45 UTC

Acronis (Static ML)	Suspicious	Ad-Aware	Trojan.GenericKD.45798479
AhnLab-V3	Malware/Win.Reputation.C4734607	Alibaba	Malware:Win32/km_283b8874.None
ALYac	Trojan.GenericKD.45798479	Antiy-AVL	Trojan/Generic.ASCommon.200
Avast	Win32:Malware-gen	AVG	Win32:Malware-gen
Avira (no cloud)	TR/Dropper.Gen	BitDefender	Trojan.GenericKD.45798479
BitDefenderTheta	AI:Packer.801320CE1E	Bkav Pro	W32.AIDetect.malware1
ClamAV	Win.Dropper.Fileinfector-9832222-0	CrowdStrike Falcon	Win/malicious_confidence_100% (W)
Cybereason	Malicious.1b56f6	Cylance	Unsafe
Cynet	Malicious (score: 100)	Cyren	W32/Agent.AGA.gen!Eldorado
DrWeb	Trojan.Click3.29339	Elastic	Malicious (moderate Confidence)
Emsisoft	Trojan.GenericKD.45798479 (B)	eScan	Trojan.GenericKD.45798479
ESET-NOD32	A Variant Of Win32/Agent.SNX	Fortinet	W32/Agent.E970!tr
GData	Win32.Trojan.PSE.RP7DJJ	Gridinsoft	Ransom.Win32.Wacatac.oa!s2
Ikarus	Trojan.Agent	Jiangmin	Trojan/Genome.cae

txOne networks

Outline

- Introduction
  - Threat Overview
  - The Difficult Problem of Static/Dynamic Malware Detection and Classification
- Deep Dive into Our Practical Symbolic Engine
  - Related Work
  - Our Practical Symbolic Engine
- Demonstration
  - CRC32 & DLL ReflectiveLoader
  - Process Hollowing
  - Ransomware Detection
- **Future Works and Closing Remarks**

Sound Bytes

- In-depth understanding of the **limitations and common issues** with current static, dynamic and machine learning detection
- In-depth understanding of **why and how we choose symbolic execution** and various auxiliary methods to build symbolic engine and learn how to create the signature to detect the kinds of attack and technique
- From our demonstration and comparison, learn that our novel method and engine are indeed superior to the previous methods in terms of **accuracy** and **validity** and can be used in the real world.
- Know the plan about opensource to gather the **community** power to strength the engine and signature

Opensource to Infosec Community

TCSA v1

TXOne Code Semantics Analyzer by TXOne Networks, inc.

Hightlight Features

1. Malware Detection, e.g. Process Hollowing & Ransomware
2. Vulnerability Scanning e.g. Firmware Command Injection
3. (unpractical) ML for Clustering Malware e.g. Neural Networks

Installation

1. Script Usage: `$pip install vivisect` then `$python3 Akali/akali.py samples/hello_recur.exe`
2. Standalone Build: `$pyinstaller .github\pyinstaller\akali.spec` then `$dist\akali.exe samples\hello_recur.exe`

https://github.com/TXOne-Networks/TCSA

txOne networks

Sheng-Hao Ma
@aaaddress1

Mars Cheng
@marscheng_

Hank Chen
@hank0438

Copyright 2022 TXOne Networks