Seance: Divination of Tool-Breaking Changes in Forensically Important Binaries

By:

Ryan Maggio (Louisiana State University),

Andrew Case (Volatility Foundation),

Aisha Ali-Gombe (Towson University),

and Golden G. Richard III (Louisiana State University)

From the proceedings of

The Digital Forensic Research Conference

DFRWS USA 2021

July 12-15, 2021

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2021 USA - Proceedings of the Twenty First Annual DFRWS USA

Seance: Divination of tool-breaking changes in forensically important binaries

Ryan D. Maggio [b, c, *], Andrew Case [a], Aisha Ali-Gombe [d], Golden G. Richard III [b, c]

[a] *Volatility Foundation, USA*
[b] *Center for Computation and Technology, Louisiana State University, USA*
[c] *School of Electrical Engineering & Computer Science, Louisiana State University, USA*
[d] *Department of Computer and Information Science, Towson University, USA*

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Program analysis
Digital forensics

ABSTRACT

The value of memory analysis during digital forensics, incident response, and malware investigations has been realized for over a decade. The power of memory forensics is based on the fact that volatile memory contains a substantial number of artifacts that are simply never recorded to disk or sent across the network in plaintext form. Orderly recovery of this data, known as structured analysis, allows for recovery of the full system state at the time of acquisition. For structured analysis to be successful, a memory analysis framework must have an accurate model of the data structures and algorithms of the target operating system and applications. Unfortunately, acquiring this layout is often a difficult task for even one version of an executable module, and the problem is only compounded when support for a wide variety of versions is desired. This issue can be manifested in several ways, including forensics frameworks being unable to process memory samples containing unsupported versions of executable code or worse, generating erroneous or incomplete results. Given the vital role memory analysis plays in modern investigations, these issues are unacceptable. In this paper, we present Seance, a system that implements automated binary analysis to provide accurate data structure layout information for different versions of targeted executed modules. The results of Seance can be consumed by analysis frameworks to accurately support all versions of a target module.

1. Introduction

The necessity of memory analysis during modern digital forensics and incident response investigations is well known and documented. Through analysis of volatile memory, investigators can recover the entire system state at the time of acquisition and deeply examine historical activity. These capabilities go well beyond the types of evidence made available by more traditional forensic analysis techniques, such as disk image forensics or live system analysis.

The importance of memory forensics has only been strengthened over time as malware and attack toolkits have evolved to use completely memory-only payloads, or at a minimum, significant memory-only portions (PowerShell Empire, 2016; hdm, 2020; GhostPack, 2020; Mudge, 2020; FireEye, 2020; Arghire, 2019). Beyond malware analysis, the now ubiquitous use of encrypted file storage and network sessions has driven the need for memory forensics.

To perform memory forensics beyond basic techniques, such as examining strings or running Yara rules, investigators must rely on memory forensic frameworks that are capable of automatically reconstructing complete system state. Widely used frameworks, such as Volatility (The Volatility Framework:, 2017) and, until recently, Rekall (Google, 2016), provide dozens of plugins that automatically reconstruct specific artifacts, such as the list of running processes, memory ranges that contain injected code, or kernel regions that have been maliciously altered.

Structured analysis is the automated recovery of data structures and algorithms and its success depends on several factors. The first is the ability to locate the address of global variables containing key

* Corresponding author. Center for Computation and Technology, Louisiana State University, USA.
E-mail address: ryandmaggio@gmail.com (R.D. Maggio).

https://doi.org/10.1016/j.fsidi.2021.301189

information, such as the head of the process list or the base address of the system call table array. As discussed later, gathering this information is generally not an issue for modern frameworks and is performed accurately.

The second piece of information required is the layout of the data structures that hold desired artifacts, including the offset and type of each member needed for complete analysis. Unfortunately, recovery of this information is often not straightforward and current workflows require intensive, manual methods to recover the information for every version of an application or operating system to be analyzed. Furthermore, there are currently no robust, automated methods to determine if the data at a given offset has the same semantics as it did for other versions of the same module. As showcased in Section 3, these issues have led to significant deficiencies in real-world analysis, including both the inability of memory forensics tools to analyze some versions of an executable module and incomplete or incorrect results being reported.

In this paper, we document our effort, called Seance, to transform this manual and error-prone process into an automated, streamlined, and verifiable workflow. To accomplish this, we developed a system capable of automatically assessing executable modules needed for structured analysis. This assessment begins with binary analysis to determine the layout of needed data structures contained within an analyzed module. A second component then allows comparing the assessment to other versions of the module to determine if the data structure layout changed, or if the code accessing structure members has materially changed. This output can then be used to determine if a memory forensic framework is currently capable of analyzing a particular module version.

The use of our system by memory forensic developers and practitioners will significantly advance the state of the art by providing for rapid development of their framework updates along with verification that data structures are represented correctly. This will greatly enhance the accuracy and reliability of frameworks through replacement of manual, error-prone efforts with an automated workflow.

2. Related work

2.1. Data structure reconstruction

For the same reasons that motivated this paper, there has been significant interest in the recovery of the data structure layout of executable modules analyzed during memory analysis investigations.

2.1.1. Linux kernel analysis

Previously (Case et al., 2010), described an effort that recovered the offset of several data structure members needed for analysis of Linux processes. This work was groundbreaking, but the techniques that were used are fragile. For instance, instead of performing full binary analysis, pattern matching was used against the disassembly of functions that access members of interest in a structure. As described in the paper, this approach can only support a limited range of kernel versions. A very closely related approach to reconstruct a subset of Linux kernel data structure layouts is taken in (Wang et al., 2016).

In (Socała and Cohen, 2016), members of the Rekall development team added support for automatic profile generation from live Linux systems. Prior efforts required the installation of compiler tools on target systems, along with other dependencies. This has obvious negative forensic impacts, so removing that requirement was desirable. Their approach involved pre-compiling ASTs for mainline kernel versions followed by runtime refinement based on the configuration of the kernel being analyzed. This allowed the creation of Rekall profiles capable of analyzing live systems without the need for compiler tools on the target system.

Additionally (Pagani, 2019), describes a system to automatically build Volatility profiles for analysis of Linux memory samples. This project uses a custom *Clang* plugin to generate kernel source code information and then uses *angr* to perform symbolic execution against functions in a memory sample. The analysis of these functions reveals the offset of structure members. *angr* is the standard open-source framework for binary analysis, and as discussed later, is used in our project as well. This project differs from ours in key ways though, including 1) it requires the source code of the target module and 2) it is unable to internally detect when its view of a data structure layout is incorrect. The authors also document several instances where Volatility plugins do not produce correct artifacts when using profiles built with their system.

2.1.2. Windows kernel analysis

The initial approaches to Windows memory analysis relied on the debugging information of the kernel executable provided by Microsoft (Petroni et al., 2006; Peterson and Okolica, 2010). This debugging information is contained within per-executable-version PDB files hosted on Microsoft's symbol server. For kernel executables, the PDB will contain the address of all global symbols as well as the layout of each data structure. This satisfies the requirements necessary for structured analysis. As noted in several places, however, there are critical memory artifacts that are not contained within the base kernel module (Ligh et al., 2014; Cohen, 2015; OMFW, 2012). Most notably are those of associated with the GUI subsystem (win32 k*.sys), the network stack (tcpip.sys), and the web server stack (HTTP.sys). For the network and web server stack, the released PDB files have only ever contained the address of global symbols. For the GUI subsystem, PDBs with data structures included were released for a small number of Windows 7 versions, while the rest have only included symbol addresses. These gaps have led to the requirement for significant, manual reverse engineering efforts on the part of memory analysis framework developers.

An approach to solving this issue was documented in (Cohen, 2015). It aimed to add support for a wide range of win32 k.sys versions within Rekall. The approach taken was to determine, through manual analysis, which functions referenced members of data structures needed for analysis to succeed. The result was the creation of template files that encode the instructions used to access a particular structure member (The Rekall Team, 2014). To provide a degree of flexibility, the template format wildcards the offsets of control flow redirecting instructions as well as the particular general-purpose register used to store and manipulate values. Unfortunately, this approach is limited in several key ways. First, as noted in the paper, the templates are fragile when it comes to changes in the compiled instruction flow. While more flexible than work before it, the templates require the specific sequence of instructions leading to a member access. This reduces usability across versions of a module, as these frequently change. Second, as discussed in Section 6, not all accesses to needed structure members occur at the beginning of a function or within a relatively small function. Since the templates require the specific ordering of instructions leading to an access (post—branch unrolling), they are fragile relative to the template's size.

To alleviate the issues of instruction matching, Seance employs advanced binary analysis that understands the semantics of structure member accesses. This allows it to determine both if the analyzed function has substantial changes between versions as well as if the needed member offset(s) have changed.

2.1.3. Virtual machine introspection

Virtual machine introspection (VMI) is a technique for memory analysis of virtual machine guests from the host. To perform this analysis, VMI software must meet the same requirements as traditional structured memory analysis software. Popular VMI methods to obtain data structure layouts, or to avoid the issue completely, include graph-based analysis, machine learning, and code-reuse (Inoue et al., 2011; Fu and Lin, 2012; Kumara and Jaidhar, 2018; Saberi et al., 2014;: HYPERSHELL; Dolan-GavittTim Leek et al., 2011). Unfortunately, none of these solves the current problems of memory analysis frameworks as the approaches either require source code access, or they simply borrow and/or re-execute running code inside of a guest to extract information from an API.

2.2. Program analysis in security

Program analysis, and specifically binary analysis, has long been a focus of security research. Projects such as BitBlaze (Song et al., 2008) and angr (; Yan et al., 2016; Stephens et al., 2016; Yan et al., 2015) provide platforms for performing these types of analysis. Analysis of control flow graphs (CFGs) is also a common technique in malware analysis and defensive security (Nguyen et al., 2018; Bruschi et al., 2006; CesareYang, 2010; Cheng et al., 2014; Yan et al., 2016). Symbolic execution has been used in numerous research efforts to automatically discover vulnerabilities, generate patches, refine fuzzing efforts, and direct coverage of binary code paths (Cadar et al., 2006; Davidson et al., 2013; WangTao et al., 2009; Yang et al., 2006, 2013; MolnarXue and David, 2009; Kolbitsch et al., 2012; Stephens et al., 2016; Chen et al., 2013; Wang et al., 2010; Dolan-Gavitt et al., 2016). The power of symbolic execution is based on its ability to automatically explore multiple (or all) code paths of a program.

3. Reconstructing data structure layouts

In this section, we document the current approaches to data structure layout reconstruction used in memory analysis frameworks, along with the limitations to these approaches. This topic was partially discussed in the related work section, but given its complexity and central role in motivating our research effort, we also have included this section for completeness.

When a memory forensic developer wishes to learn the layout of a data structure of interest, there are three possibilities (Ligh et al., 2014). The first two, source code review and use of debugging symbols, are generally easier than the third, binary analysis, but as discussed, are not always accurate, complete, or even feasible. In those situations, binary analysis is the only choice available. Luckily, if an application accesses target members of a data structure, those accesses will be encoded within instructions in the executable module. This means binary analysis is not only possible in virtually every context, but it is also guaranteed to encode the correct offset.

3.1. Source code review

3.1.1. Approach

This method uses access to the source code of an application to determine the name, layout, and purpose of data structures of interest. By knowing the compiler rules for how data structures are laid out in memory compared to their source code representation, is it often possible to manually build out the data structure layout.

3.1.2. Limitations

The first limitation of this approach is that it is extremely time-consuming and error-prone for large code bases. One miscalculation for a structure member will break the derived offset for all remaining members. Considering dozens of data structures are often needed for deep analysis, and that each data structure often has dozens of members, the chance of such miscalculation occurring is high. Furthermore, to support the 32-bit and 64-bit versions of applications, this process must be repeated twice for each module release. Also complicating matters is that the compiler and linker optimizations can create significant changes in the data structures and ultimately the final view of the binary. As discussed in related works (Case et al., 2010; Pagani, 2019; Cohen, 2015), compiler alignment of structure members have a drastic effect on this layout, and the alignment choices are not always discernible just from viewing the source code.

3.2. Use of debugging symbols

3.2.1. Approach

When debug files contain complete type and symbol information then nothing else must be done except to convert that information into a representation that the memory analysis framework understands. The two main file formats for this information are PDB files from Visual Studio and DWARF files on Linux and Mac. The PDB file format has been reverse engineered to enable extracting all information and the DWARF format is fully documented.

3.2.2. Limitations

The main limitation of this approach is that complete debugging information is not published for many of the modules needed for memory analysis. As mentioned previously, some vendors publish debug information of their production executable modules, and the two most commonly used in memory forensics are the Microsoft symbol server and the kernel build repositories maintained by various Linux distributions. Unfortunately, those published by Microsoft are usually only complete for the kernel executable itself.

In the situation where the source code of a target module is available, but debug files weren't published by the vendor, researchers will often compile the source code on their own with debugging enabled to generate a local debug information file. When the compiler version and settings of the target module are known then this workflow is generally accurate, but accuracy decreases when the exact build environment configuration cannot be replicated. Furthermore, there are platforms, such as macOS, that mixed closed and open-source code, so compiling the open source code is not always possible with the absence of code from closed source components.

3.3. Binary analysis

3.3.1. Approach

Given the inability of the previous methods to work in many situations, memory analysis capabilities are often built using binary analysis. This approach requires an expert investigator to find the functions within a module that reference needed structure members.

3.3.2. Limitations

This is a time consuming approach that requires an expert to manually reverse engineer many components of target modules. The expert must then manually record all offsets and types discovered into the target framework's format. This makes the effort not only extremely time consuming, but also error-prone and non-repeatable.

3.4. Shared limitations

There is also a significant, shared limitation of all three approaches in that they only attempt to acquire the offsets of needed members of a data structure. As demonstrated with Objective-C in Section 5, when attempting to correctly support a wide variety of versions, just having the offset is not enough. Instead, the *semantics* of the operations performed on that offset must be understood, so that frameworks can find and report valid artifacts. The combined issues and limitations of current approaches all necessitate the need for automated approaches to processing modules at the instruction level.

4. Automated analysis of data structures across program versions

Seance leverages angr (Yan et al., 2016; Stephens et al., 2016; Yan et al., 2015; http://angr.io/), a robust binary analysis platform, to perform an automated analysis of the target module. angr provides a wide array of features and modes, and the subset we utilized includes:

- Loading binaries
- Scanning loaded binaries
- Locating supported symbols
- Performing symbolic execution of those symbols
- Instrumenting the symbolic execution to collect data
- Producing a control flow graph of the symbols

After the data is collected, we process it, identifying any relevant features and condensing them into a fingerprint. Fingerprints produced by Seance are then added to a database of fingerprints for previously examined versions of the same code and also tested against the database to check for differences. If any differences are found between versions in the database, a short description of the nature of the discrepancy is generated. The rest of this section describes this workflow in detail.

4.1. Structure offset access specification

Previous efforts to automatically determine the structure layout largely relied on small functions in target binaries that accessed needed structure members. This was necessary as both static and brittle nature of the signatures, or templates used in these systems could not handle variability in instructions and/or the registers used in member accesses. Unfortunately, as discussed in Section 6.5, target applications often only reference members in large and complex functions. Furthermore, when binary analysis is required to determine functions that make accesses, it is not always feasible to find small, easy to parse functions, if they are present at all.

For these reasons, Seance provides several methods to generically specify which functions access data, and, optionally, how that access occurs. The most basic method is for functions that receive as a parameter the data structure under analysis and that access one or more needed members of that structure. Seance then only requires the function name and parameter number of the data structure to track offset-based accesses to it. This simple specification allowed recovery of all required Objective-C members, as discussed in Section 5.

For more fine-grained control over what operations require human post-processing review, more detailed specifications can be given. The first option is to specify that the offset(s) of interest are used to store the return value of a function call made inside the analyzed function. The second is to specify that the offset(s) of interest are passed as specific parameters to a function called by the function being analyzed. This precise specification allows Seance to not only detect changes in the algorithms of the analyzed module via CFG analysis but to also ensure that the precise operations used to calculate artifacts remain constant. This provides a high level of assurance that memory analysis frameworks are recovering expected data. Examples of using these precise specification methods are shown in Section 6.5.

4.2. Data collection

Using angr's binary loader, generally referred to as the CLE, the target binary is loaded into a virtual address space initialized by angr. Before it can begin CFG generation, angr requires the starting and ending address of the code to analyze. The starting address is specified by the controlling script as either an offset from the target module or as a symbol name. To find a reasonable value for the ending address, Seance first scans the target module for a symbol with a starting address closest to, but larger than, the target symbol's starting address. Then, Seance scans backwards from that address looking for either a *ret* or *int3* instruction. These are the most common markers of a function's end as the *ret* instruction is used to return control flow to the previously executing function and *int3* instructions, which force a debug trap, are placed by the compiler to catch programming errors. This scan is not perfect, as it might not capture all the ways the function might exit, but it provides reasonable bounds.

To begin CFG generation, we feed the starting address and calculated ending address to angr's CFGFast function (angr, 2021). This function receives a starting and ending address tuple, performs control flow graph generation, and then produces a list of termination addresses along with metadata for each graph node. As a sanity check, we verify that our calculated ending address is included in angr's produced list of termination addresses. The ending addresses, along with the other metadata, including the number of nodes and leaves that appear in the graph, as well as how many functions are called, are then stored in a JSON file. These features are later used as a heuristic for comparing CFGs between versions of analyzed modules.

Next, we perform symbolic execution passes with each of the ending addresses as an execution's terminating address. By default, each register and memory range is initialized as zero. To track access to registers and memory, Seance leverages callbacks provided by angr for monitoring reads and writes to both data sources.

For each access, the callbacks receive:

- The address (or register) being accessed
- The data being handled
- The length of the access, in bytes
- The condition under which the access occurs
- The address of the block in which the access occurred

These data are stored in one of two dictionaries (one for reads, and one for writes), to be processed later.

For symbolic execution, we use angr's built-in *explore* exploration technique, targeting each of the possible ending addresses we discovered.

After execution terminates, we process the data collected during execution, as described in the next section. Lastly, we generate another CFG, constrained to only touch basic blocks which were discovered during symbolic execution. This CFG is used as a sanity check on our results.

4.3. Analysis results database

The results of Seance's automated analysis are stored in a JSON file containing the destination or source of each memory access, along with a concretization of the value stored or read. This data is recorded from each round of symbolic execution, which tells us how memory was accessed over several different possible execution paths and affords us a much more complete idea of how the function uses memory.

Once the processing of memory accesses is complete, we find the initial value held by each register and record the offset of each memory access from that value. If the register holds a pointer to a data structure of interest, these offsets tell us the locations of each member of the data structure accessed by the function.

We repeat this process separately for memory addresses which were read from, instead of the initial values of registers. This gives us a list of pointers accessed from memory, and offsets from that pointer which were also accessed. This data is further cross-referenced with the list of memory reads and the list of offsets to trace the origins of the pointers in memory.

We treat this as a tuple whose first element is the source, and whose second element is an offset-list, and store it as an entry in a dictionary, which is then stored in a JSON file. Reading out this data produces output as seen in Fig. 1. Once data has been collected for all desired versions of some binary, it is consolidated into a database for ease of analysis. Specifically, the database is a JSON file with each entry a dictionary with the following information:

- File version
- List of source/offset—list pairs
- Number of nodes, leaves, and functions in the CFG

This database can then serve as the point of comparison for all future versions of the software we wish to analyse.

4.4. Comparing results

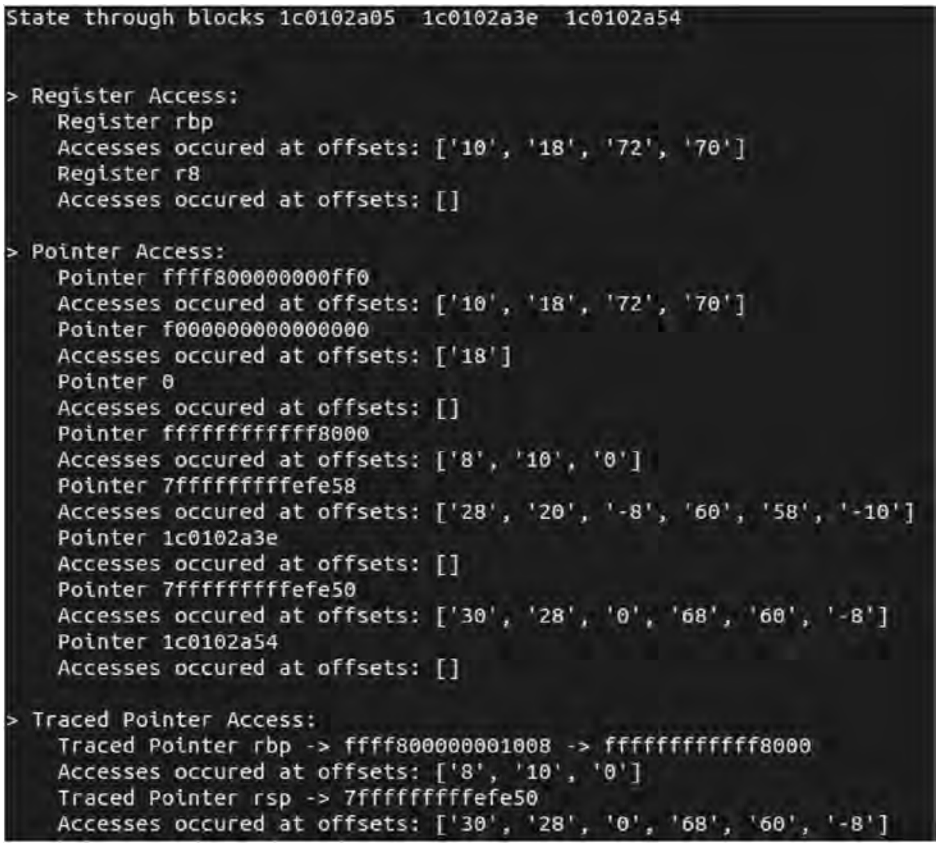
A powerful component of Seance allows comparing the results of analyses across different versions of a target executable. This comparison is then used to inform developers and investigators of the suitability of an analysis framework to analyze a specific module version.

The comparison across versions is driven by the analysis of the CFG metadata as well as the offset-based accesses found during symbolic execution. The CFG is compared to determine if it exactly matches across versions, meaning if the number of nodes and leaves are the same, as well as the number of external functions called.

The offset-accesses are compared multiple ways to gauge levels of similarity. First, for each entry in the database, we perform a literal comparison, noting the software version on a match. If the match is not identical, that is if each offset in the new version does not have exactly the same associated list of offsets as in the old version, then we perform a more fine-grained comparison. In this case, we remove duplicate accesses in the old and new versions and perform another literal comparison, adding the old version to a separate list if it matches.

If this more focused comparison does not match, we then compare the stripped offset lists on an access-by-access basis, checking if any of the accesses had the same offsets referenced between versions. If any of the accesses match here, the list of matching accesses is added to a new list along with those from the old version.

The direct comparison of CFG data, along with three levels of access checks, leave us with five outcomes:



```
State through blocks 1c0102a05  1c0102a3e  1c0102a54


> Register Access:
    Register rbp
    Accesses occured at offsets: ['10', '18', '72', '70']
    Register r8
    Accesses occured at offsets: []

> Pointer Access:
    Pointer ffff800000000ff0
    Accesses occured at offsets: ['10', '18', '72', '70']
    Pointer f000000000000000
    Accesses occured at offsets: ['18']
    Pointer 0
    Accesses occured at offsets: []
    Pointer ffffffffffff8000
    Accesses occured at offsets: ['8', '10', '0']
    Pointer 7ffffffffefe58
    Accesses occured at offsets: ['28', '20', '-8', '60', '58', '-10']
    Pointer 1c0102a3e
    Accesses occured at offsets: []
    Pointer 7ffffffffefe50
    Accesses occured at offsets: ['30', '28', '0', '68', '60', '-8']
    Pointer 1c0102a54
    Accesses occured at offsets: []

> Traced Pointer Access:
    Traced Pointer rbp -> ffff800000001008 -> ffffffffffff8000
    Accesses occured at offsets: ['8', '10', '0']
    Traced Pointer rsp -> 7ffffffffefe50
    Accesses occured at offsets: ['30', '28', '0', '68', '60', '-8']
```

Fig. 1. Reading memory accesses back out of JSON files generated from analysis of a function in Windows' tcpip.sys.

1. Full fingerprint match, meaning the CFG and offset accesses were the same
2. Offset match with CFG change, meaning the offsets were the same but the code changed
3. Raw offset match, meaning the offsets were the same but their ordering and/or number of accesses changed
4. Specific access match, meaning only a subset of accesses in the list matched
5. No matches, meaning the CFG changed and no previously found offsets were matched

This presentation of results quickly tells investigators if their tools will work as intended or if a change is needed. It also informs developers as to which action(s) must be taken to support the new version.

Outcome 1) requires no changes to the framework, 2) and 5) indicate that the targeted function should be re-analyzed to ensure consistency of extracted artifacts, 3) generally occurs as a result of compiler code re-ordering between versions and does not require re-analyzing the function, and 4) requires re-analysis only if offsets relevant to the framework have changed.

4.5. *Advantages of seance*

As documented in this section, the automatic binary analysis capabilities of Seance allow investigators to rapidly determine if a memory analysis framework supports a specific version of a target module. Seance also provides a capability for generic specifications that describe access to a particular offset or set of offsets. These specifications are not brittle to instruction changes, register changes, or even substantial compiler code re-ordering as Seance's semantic analysis is driven at a higher level through symbolic execution. Overall, these features will allow for a significant shift in the reliability and speed in which memory analysis frameworks can support a wide range of operating system and application versions. The following two sections showcase these capabilities against the macOS userland runtime, Objective-C, as well as the Windows networking stack.

5. Evaluation - Objective C runtime

5.1. *Motivation and history*

Our first full evaluation of Seance was conducted against the Objective-C runtime. We chose this as our target for several reasons, including that is commonly abused by malware on macOS systems, its sole previous research effort is largely outdated, and that it is open source.

In 2016, a paper was published at DFRWS that documented memory forensic algorithms for detecting the Crisis malware and its abuse of the Objective-C runtime (Case and Richard, 2016). This was accomplished through enumeration of Objective-C classes and data loaded into a process followed by checking for signs of malicious activity. This work only targeted macOS version 10.9, however, which stopped being supported in 2016 by Apple and is now six major OS versions of out date. When our team attempted to re-use the algorithms described in this paper against modern Objective-C runtimes, many parts of our plugin completely broke down.

The realization that many newer versions would need to be supported, all requiring manual verification and testing, was daunting. We also knew that the memory forensics community as a whole was facing similar issues with applications and modules across a number of other operating systems. We then decided that an automated system for determining version compatibility of memory forensic algorithms was needed, and, hence, Seance was born.

5.2. *Targeted data structures*

Our goal for revamping Objective-C analysis was to be able to gather all loaded classes in a process address space followed by the enumeration of instance variables and methods for each class. This information allows us to enumerate active instances of each class and decode the values of variables. Table 1 lists the structures and members needed to accomplish this, along with the functions that reference each member.

5.3. *Versions tested*

We conducted our tests using versions of the Objective-C runtime corresponding to macOS versions 10.11.0 through 10.15.6. To gather the versions needed for testing, we first extracted the *libobjc.dylib* from each of these macOS versions. We then de-duplicated based on the SHA1 hash of the library and stored which Objective-C versions mapped to each hash. In total, this set was comprised of 21 different files.

5.4. *Methodology*

We attempted to simulate a realistic investigative scenario in our testing. To accomplish this, we constructed a Seance database from a selection of fourteen of the files, as described in Section 4.3. We wanted a database that might reflect the experiences of a seasoned investigator, and so opted to include more files in the database than not, and since investigators do not typically get to choose which machines they analyze, random chance was used to select the files included in the database. Once the database was constructed, each of the files left out of the database, our experimental group, were compared against the database as described in Section 4.4 for each of our target functions.

5.5. *Analysis results*

To analyze the results, we first looked at the differences detected as compared to the chronology of the files. As expected, files which Seance registered as identical matches or identical matches with a CFG change for a given function, formed contiguous groups across Objective-C versions.

For example, NXFreeHashTable generated exact matches on three distinct groups of files, 10.13.0–10.14.3, 10.14.4–10.14.6, and 10.15.0–10.15.6, whereas NXEmptyHashTable generated matches for only one large grouping, 10.14.0–10.15.6. In contrast, in the relaxed mode where only the parameter of interest is matched, then NXEmptyHashTable matches on every file tested.

Table 1
Obj-C members targeted, their containing structures, and the functions which access that member.

Member	Structure	Function
Name	i_var	getName
Offset	i_var	getOffset
Type	i_var	getTypeEncoding
Imp	method_t	getImplementation
Name	method_t	getName
Count	NXHashTable	NXEmptyHashTable
buckets	NXHashTable	NXFreeHashTable
nbBuckets	NXHashTable	NXInitHashState
Count	NXHashTable	NXResetHashTable
Isa	objc_object	object_getClass
Bits	objc_class	removeSubclass

Similarly, NXResetHashTable matches usage of the first parameter for all versions except 10.13.0—10.13.3, which corresponds to exactly one of our test files. method_getName does not generate any additional matches under these relaxed conditions, but even with exact match enabled, only has two variants. The first covers 10.11.0—10.15.3 and the second 10.15.4—10.15.6. Interestingly, we found that only the object_getClass function actually changes its CFG without using the parameters differently. This highlights the importance of CFG matching, since an investigator must check that the data structure is still used in the same way under the new algorithm.

One particular function, removeSubclass, did not produce any exact matches, only parameter matches, but these, too, coincided with temporal boundaries. However, in this case, a pattern was not readily apparent just by looking at the results file, and required source code analysis to make sense of the differences.

This brings us to the second stage of our analysis, where we looked at the source code corresponding to each file along the temporal boundaries to verify the changes reported by Seance. At this stage, we discovered that in most cases when we detect changes in offset or CFG, there was no corresponding change to the source code. The exceptions to this were the functions object_getClass and removeSubclass. In the source file objc-object.h, we found the inlined function getIsa, which object_getClass uses, was majorly changed between OSX versions 10.11.6 and 10.12.0, and again between 10.14.6 and 10.15.0. Similarly, in the source file objc-runtime-new.mm, we found the function removeSubclass underwent major revisions between OSX versions 10.14.3 and 10.14.4, and again between 10.14.6 and 10.15.0. Finally, to verify the temporal boundaries where there was no source code change, we examined the disassembly of the functions NXEmptyHashTable, NXFreeHashTable, and NXResetHashTable, where Seance detected changes, but source code analysis revealed nothing. In each of these cases, we found semantically meaningful differences in the assembly instructions.

While studying the source code corresponding to material changes reported by Seance, we learned that the algorithm to recover the variables and methods belonging to a class had substantially changed in Objective-C for macOS 10.15.0. In particular, the bits member, which previously pointed to a class_rw_t structure, could now point to either a class_rw_t or a class_ro_t depending on the class' state. Furthermore, the class_rw_t structure itself had been broken into two separate structures. This discovery illustrates the need for data structure layout extraction processes to not only find the correct offset, but also to verify the operations performed on the offset. In this situation, the bits member is used across versions, but the type it references has changed in such a drastic manner that just knowing its offset is not enough to correctly recover artifacts. We were directly pointed to this discrepancy by Seance.

5.6. Analysis conclusions

Our evaluation of Objective-C has demonstrated that Seance is capable of analyzing a wide variety of versions of a real-world library. We initially used source code analysis to derive the function list for Seance to analyze to find members and the rest of the process was automated. The results of this automated processing alerted us to new structure offsets across versions as well as where just updating a structure's offset would not be enough to accurately support analysis. This evaluation also highlighted the pitfalls of source code-only review, including that substantial changes can happen in the compiled form, as well as the downsides of data structure layout extraction workflows that only examine changes in offsets.

6. Evaluation - Windows networking stack

6.1. Motivation and history

Analysis of the data structures of the Windows networking stack provides extremely valuable artifacts during an investigation. Through this analysis, an investigator can uncover all listening sockets and connections, network interfaces in promiscuous mode, and can map network activity to the process responsible for it. These data structures also contain timestamps that denote when specific activities started and/or ended.

As incident response investigations often start as a result of a network indicator, such as a system contacting a known-bad IP address or resolving a known-bad hostname, it is a significant advantage if the investigator can quickly determine which processes were responsible for the malicious behaviour. Importantly, uncovering the connection creation time allows including recovered connections into investigative timelines.

Unfortunately, history has shown that key parts of these data structures vary greatly between tcpip.sys versions and that frameworks have not kept up with the changes. Browsing the Volatility 2 and Rekall issue trackers finds over twenty tickets related to connections not being reported or key metadata, particularly a connection's create time and owning process, not being reported correctly. We have also experienced these issues during our own investigations.

Knowing the importance of recovering network activity from memory samples, we sought to use Seance to widen the support of tcpip.sys versions in Volatility.

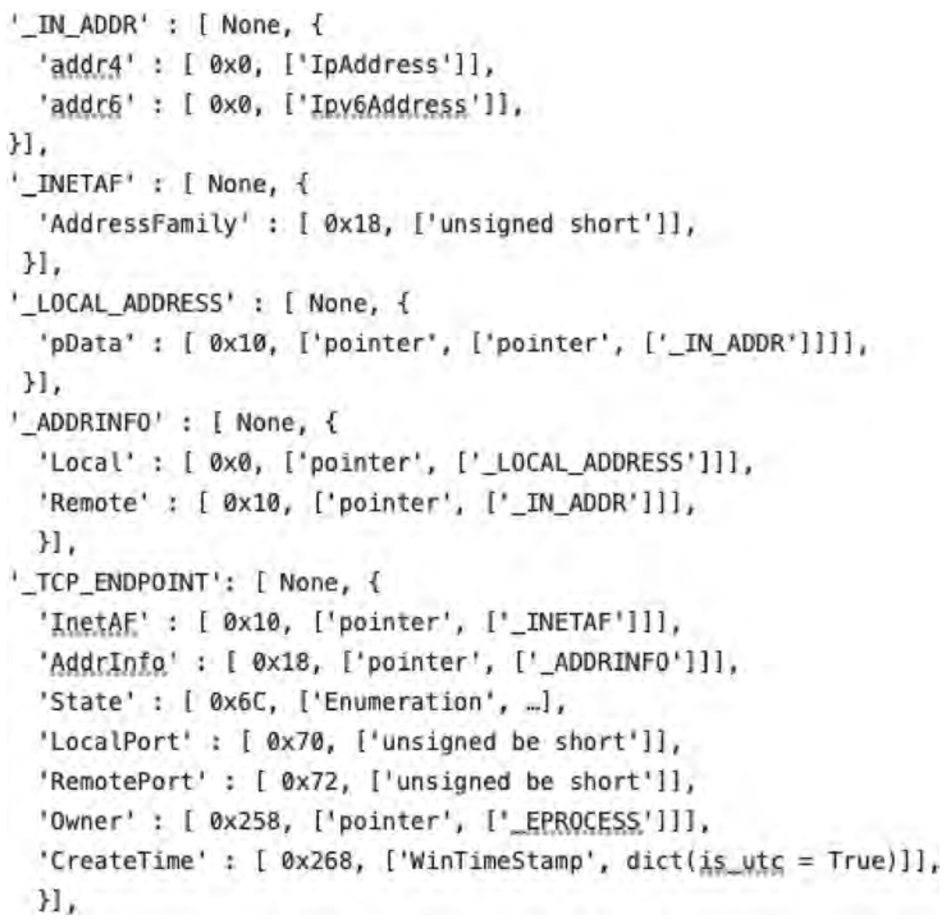
6.2. Targeted data structures

For our evaluation, we chose to target the *TCP_ENDPOINT* data structure. This structure holds the creation time, owning process, state, address family, and local and remote IP address and port for each connection. Fig. 2 displays this structure and its related-structures as defined by Volatility for the base version of Windows 10. As illustrated in the figure, TCP_ENDPOINT holds the state, port information, owner, and create time information directly in the structure. Recovery of the address family and remote and local IP addresses requires use of the related structures.

To support analysis with Seance, we needed to determine functions inside of tcpip.sys that accessed needed members. To start this process, we generated ground-truth information of the TCP_ENDPOINT structure layout for particular versions. To accomplish this, we took a two-step approach. First, we examined the source code of Volatility 2 and, later, Volatility 3 to determine the versions for which they had TCP_ENDPOINT defined. For Volatility 2, our analysis showed that the entire structure was defined for the base versions of Windows 10 and that the Owner field was updated for version 15063. No other version had updated offsets for any members.

When analyzing Volatility 3, it initially had no support. We later found a very recent effort, beginning in December 2020, to bring support to Volatility 3. From our review of this effort, the structures appear accurate from version 15063 through version 19041, but there is no support for 20H2 or for the pre-2017 Windows 10 versions. We also noted that the JSON files containing the structure layouts include a comment that they were created by hand. Furthermore, we found a few related commits indicating the structure layouts were incorrect. This is not surprising, as since manual creation of complicated structure layouts is currently a very difficult, manual task. Eliminating this manual, error-prone process was a key motivating factor for our research.

Once we observed the offsets used by Volatility 2 and 3, we then

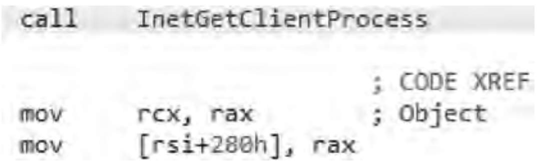


```
'_IN_ADDR' : [ None, {
  'addr4' : [ 0x0, ['IpAddress']],
  'addr6' : [ 0x0, ['Ipv6Address']],
}],
'_INETAF' : [ None, {
  'AddressFamily' : [ 0x18, ['unsigned short']],
 }],
'_LOCAL_ADDRESS' : [ None, {
  'pData' : [ 0x10, ['pointer', ['pointer', ['_IN_ADDR']]]],
 }],
'_ADDRINFO' : [ None, {
  'Local' : [ 0x0, ['pointer', ['_LOCAL_ADDRESS']]],
  'Remote' : [ 0x10, ['pointer', ['_IN_ADDR']]],
  }],
'_TCP_ENDPOINT': [ None, {
  'InetAF' : [ 0x10, ['pointer', ['_INETAF']]],
  'AddrInfo' : [ 0x18, ['pointer', ['_ADDRINFO']]],
  'State' : [ 0x6C, ['Enumeration', …],
  'LocalPort' : [ 0x70, ['unsigned be short']],
  'RemotePort' : [ 0x72, ['unsigned be short']],
  'Owner' : [ 0x258, ['pointer', ['_EPROCESS']]],
  'CreateTime' : [ 0x268, ['WinTimeStamp', dict(is_utc = True)]],
  }],
```

Fig. 2. Data structures for recovering network connections.

began an intensive binary analysis effort against tcpip.sys to discover functions that reference the needed structure offsets. Given the complexity of tcpip.sys, this was a non-trival task. Our analysis initially revealed that the State member is referenced in TcpCanTcbSend, but later testing with Seance revealed that we needed to use a different function to support Windows 10 version 20H2. This is discussed further in Section 6.5. We also determined that the Owner and CreateTime fields are accessed in TcpCreateAndConnectTcb, and that all remaining fields are accessed in TcpConnectTimeout.

We also determined that we could strengthen the checks for all members except State and CreateTime by using the advanced specification features of Seance. For the Owner member, we discovered that it was populated through a call to InetGetClientProcess. This function returns a pointer to the _EPROCESS structure of the process responsible for creating the connection. The disassembly of this flow is shown in Fig. 3. Note that the RAX register is used as the return address of 64-bit function calls. To ensure that the time value is always populated by InetGetClientProcess, we specified it as a return value check in our specification for TcpCreateandConnectTcb.



```
call    InetGetClientProcess

                        ; CODE XREF
mov     rcx, rax        ; Object
mov     [rsi+280h], rax
```

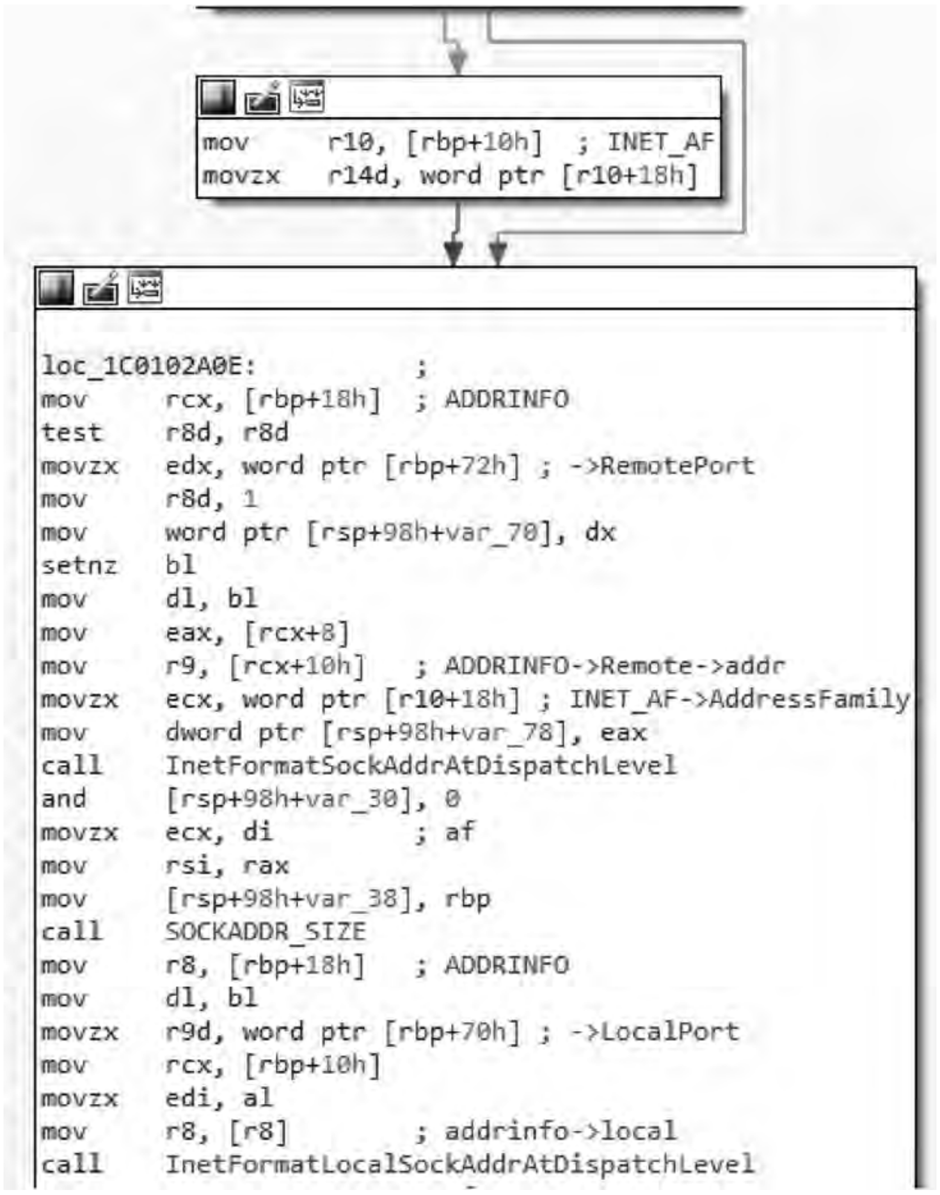
Fig. 3. Owner being assigned from InetGetClientProcess

For the members that track the address family and local and remote IP addresses and ports, we created a calling function specification. In particular, we noticed in our examination of TcpConnectTimeout that these members are passed as parameters to InetFormatSockAddrAtDispathLevel and InetFormatLocalSockAddrAtDispatchLevel. These functions are responsible for creating sockaddr_in structures based on the parameters sent. Fig. 4 shows our commented disassembly of how each member is prepared before both function calls. This allowed us to specify a function parameter check for TcpConnectTimeout.

6.3. Versions tested

To ensure that our effort covered a substantial variety of modern Windows versions, we tested every major 64-bit Windows 10 version released to date. The following list provides the OS version and corresponding build numbers covered in our testing. We note that this data set covers versions going back to the year 2016 through the latest release at the time this paper is being written.

- 1604 - 10586
- 1607 - 14393
- 1704 - 15063
- 1709 - 16299
- 1803 - 17134
- 1810 - 17763



```
mov     r10, [rbp+10h]   ; INET_AF
movzx   r14d, word ptr [r10+18h]
```

```
loc_1C0102A0E:          ;
mov     rcx, [rbp+18h]  ; ADDRINFO
test    r8d, r8d
movzx   edx, word ptr [rbp+72h] ; ->RemotePort
mov     r8d, 1
mov     word ptr [rsp+98h+var_70], dx
setnz   bl
mov     dl, bl
mov     eax, [rcx+8]
mov     r9, [rcx+10h]   ; ADDRINFO->Remote->addr
movzx   ecx, word ptr [r10+18h] ; INET_AF->AddressFamily
mov     dword ptr [rsp+98h+var_78], eax
call    InetFormatSockAddrAtDispatchLevel
and     [rsp+98h+var_30], 0
movzx   ecx, di         ; af
mov     rsi, rax
mov     [rsp+98h+var_38], rbp
call    SOCKADDR_SIZE
mov     r8, [rbp+18h]   ; ADDRINFO
mov     dl, bl
movzx   r9d, word ptr [rbp+70h] ; ->LocalPort
mov     rcx, [rbp+10h]
movzx   edi, al
mov     r8, [r8]        ; addrinfo->local
call    InetFormatLocalSockAddrAtDispatchLevel
```

Fig. 4. Several members being accessed from TcpConnectTimeout

- 1903 - 18362
- 1909 - 18363
- 20H2 - 19042

6.4. Methodology

To begin our testing process, we generated a Seance database for Windows 10 version 17134. We then compared the generated database to the information within Volatility 3. We also ran the Volatility 3 plugin to ensure connection structures were recovered, along with their complete metadata. Once confirmed as accurate between the data found during our binary analysis efforts, Volatility source code review, and Volatility plugin testing, we then used this as a comparison point for databases generated against all over tcpip.sys versions.

Creating the databases for tcpip.sys required a few extra steps, compared to our work with Objective-C. First, we needed the PDB files for each version under analysis and we also needed the symbol address and names in a parseable format. To accomplish this we relied on pdbparse, which is an open-source tool that can both download and export symbol information. Once the symbol information was available, we could then provide angr with the offsets of symbols needed for analysis.

6.5. Analysis results

The results of our analysis revealed several interesting insights. Initially, we observed two issues with version 19042. First, it reported that the symbol we had configured for recovering the TCP state, TcpCanTcbSend, did not exist in the PDB. Manual examination of the pdbparse output confirmed this. This required us to re-analyze tcpip.sys to find a function present in 19042 that accessed the offset. Our subsequent binary analysis found this access in TcpComputeRtoTcb.

The second issue reported was a material CFG change when attempting to recover the CreateTime member. This occurred as all previous versions acquired the time by dereferencing the value at the hardcoded address 0xFFFFF78000000014. This address is not documented by Microsoft, but analyzing cross-references to it gives a strong indication that it is the current system time. Starting with 19042, we found that the creation time of connections is instead calculated by calling the KeQuerySystemTimePrecise function. This significant change is what triggered the Seance report.

Looking at our results as a whole, we found that the offsets of members for tracking the connection state, family, and remote and local IP addresses and ports did not change in any version. We also noted that these members are near the beginning of the structure. We also found that the CreateTime and Owner offsets changed for *every* version analyzed, but that the offset between the two was always 16 bytes.

6.6. Analysis conclusions

Our analysis of tcpip.sys showed that even though the driver is extremely complex, Seance is able to automatically calculate and report both the offsets used to access particular members as well as detect material changes that require manual review. In total, Seance only required two changes to how particular offsets are calculated compared to our initially generated database - the State and Owner members in version 20H2, as previously described. Furthermore, the necessity for these two changes was automatically detected and reported by Seance.

7. Conclusions and future work

As showcased in our evaluations against Objective-C and the Windows networking stack, Seance is capable of replacing what is currently a manual and error-prone process with one that is automated. This automated process only requires human intervention when the code relied on for member offset extraction significantly changes. This is by design as memory analysis developers need to know when key algorithms change to ensure that their framework can produce relevant and correct data.

By relying on Seance, developers can be assured that their frameworks have the correct data structure layout and that they can rapidly support varying versions of an executable with confidence. Memory forensic investigators also benefit as they will be presented with more robust, accurate, and complete sets of results when analyzing a sample. Furthermore, the creation and integration of tools such as Seance will provide a significant example of how verifiable and repeatable procedures can be used to drive tool design in the digital forensics field. This will in turn increase the reliability and robustness of algorithms that power digital forensic analysis.

Going forward, we will more closely integrate Seance with Volatility, as it is the most widely used memory analysis framework in the field. This includes producing the vtype data needed for Volatility 2 to support all Windows 10 versions as well as automatic generation of the JSON format needed for Volatility 3. We also plan to analyze the win32 k*.sys files, with a goal of bringing GUI analysis to all versions of Windows 7 along with fresh support for Windows 8 and 10.

Acknowledgment

This research was supported by NSF awards 1703683 and 1946626.

References

angr, 2021. Cfg - angr documentation. https://docs.angr.io/built-in-analyses/cfg.
Arghire, Ionut, 2019. Russian hackers silently hit government targets for years. https://www.securityweek.com/russian-hackers-silently-hit-government-targets-years.
Bruschi, Danilo, Martignoni, Lorenzo, Monga, Mattia, 2006. Detecting self-mutating malware using control-flow graph matching. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 129–143.
Cadar, Cristian, Ganesh, Vijay, Pawlowski, Peter M., Dill, David L., Engler, Dawson R., 2006. EXE: automatically generating inputs of death. In: CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security. Alexandria, Virginia, USA, ISBN 1-59593-518-5, pp. 322–335.
Case, Andrew, Richard III, Golden G., 2016. Detecting objective-C malware through memory forensics. In: Proceedings of the 16th Annual Digital Forensics Research Workshop (DFRWS).
Case, A., Marziale, L., Richard III., Golden G., 2010. Dynamic recreation of kernel data structures for live forensics. In: Proceedings of the 10th Annual Digital Forensics Research Workshop (DFRWS).
Cesare, Silvio, Yang, Xiang, 2010. Classification of malware using structured control flow. In: Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing, vol. 107, pp. 61–70.
Chen, Ting, Zhang, Xiao song, Shi ze, Guo, yuan Li, Hong, Wu, Yue, 2013. State of the art: dynamic symbolic execution for automated test generation. Future Generat. Comput. Syst. 29 (7), 1758–1773.
Cheng, Yueqiang, Zhou, Zongwei, Yu, Miao, Ding, Xuhua, Deng, Robert H., 2014. In: ROPecker: A Generic and Practical Approach for Defending against ROP Attack.
Cohen, Michael I., 2015. Characterization of the Windows kernel version variability for accurate memory analysis. In: Proceedings of the 2015 Digital Forensics Research Workshop EU (DFRWS-EU).
Davidson, Drew, Moench, Benjamin, Ristenpart, Thomas, Jha, Somesh, August 2013. FIE on firmware: finding vulnerabilities in embedded systems using symbolic execution. In: 22nd USENIX Security Symposium (USENIX Security 13). USENIX Association, Washington, D.C., ISBN 978-1-931971-03-4, pp. 463–478. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R., 2016. LAVA: large-scale Automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 110–121.
Dolan-Gavitt, Brendan, Tim Leek, Zhivich, Michael, Giffin, Jonathon, Lee, Wenke, 2011. Virtuoso: narrowing the semantic gap in virtual machine introspection. In: 2011 IEEE Symposium on Security and Privacy. IEEE, pp. 297–312.
FireEye, 2020. Highly evasive attacker leverages SolarWinds supply chain to compromise multiple global victims with SUNBURST backdoor. https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html.
Fu, Yangchun, Lin, Zhiqiang, 2012. Space traveling across VM: automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: 2012 IEEE Symposium on Security and Privacy. IEEE, pp. 586–600.
GhostPack, 2020. GhostPack. https://github.com/GhostPack.
Google, 2016. Rekall. https://github.com/google/rekall.
hdm, 2020. Metasploit. https://www.metasploit.com/.
http://angr.io/.
HYPERSHELL: A practical hypervisor layer guest OS shellfor automated in-vm management.
Inoue, Hajime, Frank, Adelstein, Donovan, Matthew, Brueckner, Stephen, 2011. Automatically bridging the semantic gap using C interpreter. In: Proc. Of the 2011 Annual Symposium on Information Assurance, pp. 51–58.
Kolbitsch, C., Livshits, B., Zorn, B., Seifert, C., 2012. Rozzle: de-cloaking internet malware. In: 2012 IEEE Symposium on Security and Privacy, pp. 443–457. https://doi.org/10.1109/SP.2012.48.
Kumara, Ajay, Jaidhar, C.D., 2018. Automated multi-level malware detection system based on reconstructed semantic view of executables using machine learning techniques at VMM. Future Generat. Comput. Syst. 79, 431–446.
Ligh, M., Case, A., Levy, J., Walters, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. Wiley, New York.
Molnar, David, Xue, Cong Li, David, A., 2009. Wagner. Dynamic test generation to find integer bugs in X86 binary Linux programs. In: *Proceedings Of the 18th Conference On USENIX Security Symposium*, SSYM'09. USENIX Association, USA, pp. 67–82.
Mudge, Raphael, 2020. Cobalt strike. https://www.cobaltstrike.com/.
Nguyen, Minh Hai, Nguyen, Dung Le, Nguyen, Xuan Mao, Quan, Tho Thanh, 2018. Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning. Comput. Secur. 76, 128–155.
OMFW, 2012. Malware in the Windows GUI subsystem. https://volatility-labs.blogspot.com/2012/10/omfw-2012-malware-in-windows-gui.html.
Pagani, Fabio, 2019. Advances in Memory Forensics. Sorbonne Université. PhD thesis.
Peterson, Gilbert, Okolica, James, 2010. Windows operating system Agnostic

memory analysis. In: Proceedings of the 10th Annual Digital Forensics Research Workshop (DFRWS).
Petroni, Nick, Walters, Aaron, Fraser, Timothy, Arbaugh, William, 2006. FATKit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digit. Invest. 3 (4).
PowerShell Empire, 2016. Empire: building an Empire with PowerShell. https://www.powershellempire.com.
Saberi, Alireza, Fu, Yangchun, Lin, Zhiqiang, 2014. Hybrid-bridge: efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14).
Socała, Arkadiusz, Cohen, Michael, 2016. Automatic profile generation for live Linux memory analysis. Digit. Invest. 16, S11—S24. ISSN 1742-2876. URL. http://www.sciencedirect.com/science/article/pii/S1742287616000050. DFRWS 2016 Europe.
Song, Dawn, Brumley, David, Yin, Heng, Caballero, Juan, Jager, Ivan, Kang, Min Gyung, Liang, Zhenkai, James, Newsome, Poosankam, Pongsin, Saxena, Prateek, 2008. BitBlaze: a new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security, ICISS '08.
Stephens, Nick, Grosen, John, Salls, Christopher, Audrey Dutcher, Wang, Ruoyu, Corbetta, Jacopo, Yan, Shoshitaishvili, Kruegel, Christopher, Vigna, Giovanni, 2016. Driller: augmenting fuzzing through selective symbolic execution. In: NDSS.
The Rekall Team, 2014. undocumented.yaml. https://github.com/google/rekall/blob/f6dd53607c8cb7a7468a2fe9a6ad6f677f71680f/src/profiles/win32k/undocumented.yaml.
The volatility framework: volatile memory artifact extraction utility framework. https://github.com/volatilityfoundation/volatility, 2017.
Wang, T., Wei, T., Gu, G., TaintScope, W. Zou, 2010. A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy, pp. 497—512. https://doi.org/10.1109/SP.2010.37.
Wang, Tielei, Tao, Wei, Lin, Z., Zou, W., 2009. IntScope: automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In: NDSS.
Wang, L., Zhang, S., Meng, X., 2016. An adaptive approach for Linux memory analysis based on kernel code reconstruction. EURASIP J. Inf. Secur. 1, 1—13.
Yan, Shoshitaishvili, Wang, Ruoyu, Hauser, Christophe, Kruegel, Christopher, Vigna, Giovanni, 2015. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS.
Yan, Shoshitaishvili, Wang, Ruoyu, Salls, Christopher, Stephens, Nick, Polino, Mario, Audrey Dutcher, Grosen, John, Feng, Siji, Hauser, Christophe, Kruegel, Christopher, Vigna, Giovanni, 2016. SoK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy.
Yang, Junfeng, Sar, Can, Twohey, P., Cadar, C., Engler, D., 2006. Automatically generating malicious disks using symbolic execution. In: 2006 IEEE Symposium on Security and Privacy (S P'06), p. 15. https://doi.org/10.1109/SP.2006.7. —257.
Yang, Zhemin, Yang, Min, Zhang, Yuan, Gu, Guofei, Ning, Peng, Sean Wang, X., 2013. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. CCS '13 1043—1054. https://doi.org/10.1145/2508859.2516676.