Digital Investigation 22 (2017) S86–S93

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2017 USA — Proceedings of the Seventeenth Annual DFRWS USA

Gaslight: A comprehensive fuzzing architecture for memory forensics frameworks

Andrew Case [a, *], Arghya Kusum Das [b, c], Seung-Jong Park [b, c], J. (Ram) Ramanujam [b, c], 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

Keywords:
Memory forensics
Computer forensics
Memory analysis
Incident response
Malware
Fuzzing

A B S T R A C T

Memory forensics is now a standard component of digital forensic investigations and incident response handling, since memory forensic techniques are quite effective in uncovering artifacts that might be missed by traditional storage forensics or live analysis techniques. Because of the crucial role that memory forensics plays in investigations and because of the increasing use of automation of memory forensics techniques, it is imperative that these tools be resilient to memory smear and deliberate tampering. Without robust algorithms, malware may go undetected, frameworks may crash when attempting to process memory samples, and automation of memory forensics techniques is difficult. In this paper we present Gaslight, a powerful and flexible fuzz-testing architecture for stress-testing both open and closed-source memory forensics frameworks. Gaslight automatically targets critical code paths that process memory samples and mutates samples in an efficient way to reveal implementation errors. In experiments we conducted against several popular memory forensics frameworks, Gaslight revealed a number of critical previously undiscovered bugs.

© 2017 The Author(s). Published by Elsevier Ltd. on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Introduction

In recent years memory forensics has become a standard component of digital forensic investigations and incident response handling. This popularity has occurred because memory forensic algorithms can find artifacts and detect system state anomalies that would go undetected by traditional disk forensics or live analysis of a running system. Because of its power and prevalence in the industry, as well as its crucial role in investigating suspicious insiders, malware, and active attackers, it is crucial that memory forensics frameworks utilize robust algorithms that are capable of withstanding tampering by malware as well as the effects of memory smear. Without robust algorithms, malware may go undetected, frameworks may crash when attempting to process memory samples, and automation of memory forensics techniques is difficult.

Memory smear (Carvey, 2005) is a common problem when non-atomic acquisition of forensic data is performed. Although it can occur when acquiring files from the local disk of a running system, it occurs more frequently when acquiring memory from an active system. Particularly on systems under heavy load, smear can result in corruption of significant portions of a memory sample. Since the contents of memory changes as the acquisition tool runs, inconsistencies in the acquired data will be present. This can result in the hardware page tables describing a memory layout that does not match what the sample contains, and it can also result in virtual memory pointers referencing invalid data. Malware that wishes to disrupt memory analysis can also freely tamper with in-memory data. These possibilities include being able to zero memory regions, overwrite regions with random bytes, and purposely manipulate pointers and data structures to reference invalid addresses or addresses that will prevent the memory forensic algorithms from uncovering malicious components.

Incorrectly handling smear and malicious tampering can lead to many undesirable outcomes, such as the framework crashing when processing input, triggering of infinite loops, or extremely long runtimes, as well as the reporting of distorted artifacts. These conditions are often obvious when an experienced investigator is

* Corresponding author.
E-mail addresses: andrew@dfir.org (A. Case), adas7@lsu.edu (A.K. Das), sjpark@cct.lsu.edu (S.-J. Park), ram@cct.lsu.edu (J. Ramanujam), golden@cct.lsu.edu (G.G. Richard).

http://dx.doi.org/10.1016/j.diin.2017.06.011
1742-2876/© 2017 The Author(s). Published by Elsevier Ltd. on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

interacting with a memory forensics framework directly, such as when running Volatility (Foundation, 2016) or Rekall (Google et al., 2016) on the command line, but they are less obvious when the framework is used indirectly by the investigator, e.g., by an automated processing harness, such as DAMM (Marziale, 2014) or VolDiff (aim4r, 2015), or in GUI or web frontends, such as VolUtil (Breen, 2015) or Evolve (Habben, 2015). In these cases, errors produced by the library are not always obvious to the investigators, since errors may be simply written to a log file, which might be examined closely only if no results are produced. In the worst cases, the frontend or automation harness does not correctly catch exceptions or error conditions from the memory framework and the errors go silently unnoticed. All of these situations are unacceptable when performing forensic analysis that must withstand legal review as well as when hunting sophisticated attackers and malware with anti-forensics capabilities.

Remedying the previously described issues requires strenuously testing the memory parsing components of analysis frameworks for handling of edge cases and corrupt memory regions. The size of the codebase and the complexity of modern memory analysis frameworks, which can process samples from a wide variety of versions of Windows, OS X, and Linux, necessitates that this testing be automated. As an example, Volatility, one of the most widely used frameworks, contains support for four hardware architectures, four operating systems, and over 200 analysis plugins. Combined, this functionality spans over 60,000 lines of code. Manual analysis of such a large code base is error-prone and clearly does not scale. Furthermore, the code base is continuously changing and as such would need constant manual review. Focusing efforts on one framework or tool is also shortsighted as there are now numerous available frameworks, both open and closed source, and all require testing.

The term “fuzzing” refers to testing programs by generating random or semi-random input to cause programs to crash or to behave incorrectly. In this paper we describe an automated fuzzing architecture named Gaslight, which can strenuously test critical components of memory forensics frameworks. Gaslight addresses all of the previously described concerns and is very efficient in terms of both processing and disk storage requirements. Specifically, we had the following goals in mind when designing Gaslight:

- Support fuzzing of both open- and closed-source memory forensics tools, without requiring modifications to the framework itself.
- Fuzz memory forensics tools written in any programming language.
- Fuzz as quickly as possible, using all available computing resources.
- Intelligently discover and report a variety of implementation errors for memory forensics tools, including crashes, infinite loops, and resource exhaustion issues.

The following sections discuss related work, describe the implementation of Gaslight, and discuss several previously undiscovered programming bugs that Gaslight automatically uncovered in the latest versions of Volatility and Rekall. The paper concludes with a discussion of our ongoing work on improving Gaslight.

Related work

Fuzzing for Security Vulnerabilities

The idea of fuzzing applications for security vulnerabilities has a long history, dating back to 1988 when Bart Miller assigned his students the task of fuzzing UNIX programs (Miller, 1988). Since then, fuzzing has become an integral part of application security testing to find bugs and vulnerabilities that would be difficult to manually spot or for which manual analysis is not always possible or scalable (Google, 2016). The most complete fuzzer currently available is american fuzzy lop (AFL) (Zalewski, 2016a), which has been used to find numerous significant vulnerabilities in widely used applications (Zalewski, 2016b).

Unfortunately, AFL, along with other similar fuzzers, are not directly applicable to memory forensics for several reasons. First, these tools require access to the source code of tools that will be tested to instrument them for analysis. This requirement violates an important goal in the design of Gaslight, specifically, that we do not require access to nor modify the source code of memory forensics frameworks being tested.

Second, AFL mutates the entire file being tested and its documentation recommends files under 1 KB in size for performance reasons. Such limitations are obviously not feasible with memory forensics, and Gaslight not only avoids making copies of files, but also targets only the portions of a memory sample that the memory forensics framework actually processes, as we discuss in the section Fuzzer Architecture.

The last issue with AFL and other similar fuzzers is that they are geared toward targeting native code (e.g., C and C++ applications). As many digital forensics tools are written in Python, these fuzzers are not immediately usable as they would be fuzzing the Python runtime instead of the tool. There was an effort (Gaynor, 2015) to make AFL operable with Python applications, but it requires significant changes to the application being tested.

Gaslight is language-independent and efficient and is capable of fuzzing any memory forensics tool or framework.

Fuzzing forensics tools

Although not directly related to our research goals, there have been two notable efforts to incorporate fuzzing into memory forensics and one major effort to fuzz disk forensics tools.

The first of these efforts was documented by Brendan Dolan Gavitt in his paper “Robust Signatures for Kernel Data Structures” (Dolan-Gavitt et al., 2009). The purpose of Brendan's effort was to determine which members of Windows' process descriptor data structure (`EPROCESS`) were critical to system stability. To test each member, virtual machine guests running Windows XP were used and individual members were mutated. After each mutation, the running guest was monitored to determine if it remained stable, crashed, or otherwise acted undesirably. The end result of this fuzzing effort was the development of scanning signatures for memory analysis that utilized only members whose values were critical. Such signatures are extremely valuable as malware cannot trivially interfere with them while also keeping an infected system stable.

A more recent effort leveraged the same workflow as Brendan to test additional structures (Prakash et al., 2015) and explored the trustworthiness of memory forensics frameworks by determining which members of structures could be mutated while still keeping the machine stable. This is essentially the inverse of Brendan's work in that Brendan focused on finding stability-critical members. This new research also supports Linux.

Although both of these efforts involve mutating volatile memory data, they do not significantly overlap the goals of the research described in this paper. Furthermore, these previous efforts cannot easily be adapted to meet our research goals, for several reasons:

1. They do not directly test memory forensics frameworks, but instead the stability of an operating system to remain stable after data is mutated.
2. Reliance on a virtual machine for mutations is significantly slower than our architecture.

3. Since Gaslight mutates *all* the data processed by a memory forensics framework, it is not limited to only certain members of certain data structures.

At Black Hat and Defcon 2007, Newsham, Palmer, and Stamos presented their efforts in using fuzzing to break the parsers of Encase and The Sleuthkit (Newsham et al., 2007). In particular, this effort focused on fuzzing partition tables, NTFS file systems, and common file types. This work revealed multiple issues in Encase and the Sleuthkit, including infinite loops, program crashes, and memory allocation errors.

Similar to Gaslight, which has mutation sets that target memory-forensic specific issues, the 2007 effort focused on common errors in file system related parsing. This effort did not attempt to minimize the amount of disk space required for fuzzing, however, and mutated entire copies of files before processing.

Dynamic taint analysis

Similar to fuzzing, dynamic taint analysis (Newsome and Song, 2005) is able to find inputs and conditions that cause programs to behave erratically. Taint analysis is more focused than vanilla fuzzing, however, since it closely monitors programs as they execute and filters inputs to reach desired portions of a program's state. Due to its power, dynamic taint analysis is now used in nearly every commercial product that tests applications for security vulnerabilities (Edward et al., 2010).

Unfortunately, while extremely powerful, dynamic taint analysis does not meet the design goals for Gaslight for two main reasons. The first is that this analysis requires modification to the source code of any memory forensics framework except those written in C or C++. This is true as otherwise the taint analysis would be performed on the interpreter instead of the memory framework. As with fuzzing, efforts have been made to utilize dynamic taint analysis in interpreted languages like Python, but these require modification to source code of the application being tested. Conti and Russo's research effort describes this approach for Python applications (Conti and Russo, 2010).

The second reason is that, even if dynamic taint analysis could be directly performed on a wide variety of programming languages, memory forensics frameworks would still have to be modified to meet the design goals for our fuzzing architecture. This is because dynamic taint analysis engines mark (meaning *taint*) untrusted inputs, such as those coming from a user or the network, to test them. Trusted inputs are not tested. Because of this design, without modification, all portions of a memory forensics framework driven by the user, such as parsing command line options, reading of environment variables and configuration files, and so on would be tested. This violates our design goal of only testing the portions of memory frameworks that actually process the memory sample.

Fuzzer Architecture

In this section we elaborate on the design goals and implementation of Gaslight.

Goals

Gaslight was designed to achieve the following goals:

- Seamless support of any memory forensics framework, whether open or closed-source, without modification to the framework. This allows for wide coverage of all existing and future memory forensics frameworks.
- Automatic scaling to utilize all available cores. Since Gaslight can generate millions of fuzzing states, it is imperative that its processing fully utilizes all available resources. In the Conclusions and Future Work section we discuss current efforts to also support distributed fuzzing.
- Perform fuzzing operations only against the portions of tested frameworks that directly process memory samples. We are not interested in fuzzing UI elements, command line parsing, and so on. The reasoning behind this decision is to concentrate on testing the portions of a framework under an attacker's control as well as code that will be utilized by automated frameworks.
- Only mutate the portions of a memory sample actually processed by the targeted framework. Since memory samples are often very large, this improves efficiency and ensures that mutations applied to a memory sample are actually processed by the analysis framework.
- Apply fuzzing mutations dynamically at runtime to avoid creating extra copies of a memory sample. Again, since memory samples are often many gigabytes in size, this requirement ensures that only the original copy of the sample is needed, and not thousands or hundreds of thousands of duplicates, each of which might have only a few bytes changed.
- Support automated identification of the following error conditions caused through fuzzing:
  - Inelegant crashes of the memory forensics framework.
  - Infinite loops or excessively long run times when processing a memory sample.
  - File-system resource exhaustion (i.e., the memory framework writing a large amount of data to the local disk due to insufficient sanity checking of processed data).
  - Memory resource exhaustion (i.e., the framework consuming an excessive amount of memory, which might affect stability).

Design and implementation

To achieve the previously described goals, the architecture depicted in Fig. 1 was designed and implemented. The main component is the fuzzing harness, which is responsible for creating and managing fuzzing tasks. This includes generating the mutations to be tested, running the tested memory forensics framework with the mutations active, and monitoring for long running tasks. The harness is currently written in Python, and is able to fully utilize all cores of the local system. Fully distributed operation is the subject of ongoing work.

Harness setup

To configure the harness, the user must specify which directories to use for storing results and temporary FUSE mount points, which plugin(s) to run, and which memory samples to mutate.

Generating fuzzing states

The first job of the fuzzing harness is to generate the fuzzing states that will be applied to the memory sample. These mutations are stored in a queue that is then processed by the fuzzing algorithm. To determine how many states to generate for a given plugin, the harness must know how many times the plugin reads from the memory sample. This is required for the fuzzer to activate mutations upon each read, as described in more detail in the following section. To accomplish this, each plugin is first run through a custom FUSE file system implementation that simply counts the number of read operations performed on the file descriptor for the memory sample. The number of reads is then saved into the results directory.

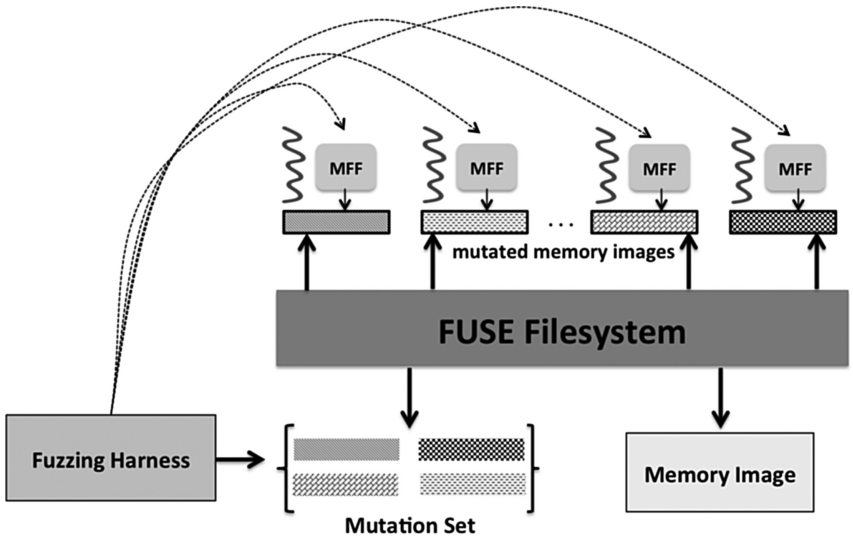

Fig. 1. High-level architecture of Gaslight. The fuzzing harness emits mutation sets, which define modifications to a memory sample that simulate smearing and malicious tampering. A custom FUSE filesystem exposes versions of the original memory sample with these mutations applied. The use of a virtual, FUSE-based file system allows Gaslight to present a mutated memory sample to an instance of a memory forensics framework (MFF in the diagram), without copying the original sample. As illustrated in the diagram, the harness spawns multiple instances of the memory forensics framework being tested in parallel to increase performance.

Once the number of read operations required for a plugin is known, generation of the fuzzing states for a plugin is accomplished with the following algorithm, incorporated into our custom file system implemented in pyfuse (Conti and Russo, 2013).

```
1 for fuzz_past in [0, 1]:
2   for read_number in range(number_of_reads):
3     for mutation_index in
          range(len(mutations.mutations)):
4       data_set = (sample, profile, plugin, fuzz_past,
                    read_number, mutation_index)
5       wq.put_nowait(data_set)
```

On Line 1, the loop is started with each possible value (0 or 1) for the *fuzz_past* variable. This variable specifies whether or not to perform mutations past the *read_number* variable set on Line 2. This is the core of the fuzzing logic as when *fuzz_past* is disabled, the fuzzer mimics one smear or malicious change to memory during one of the plugins reads. If *fuzz_past* is enabled, then it will mutate on every read starting with *read_number*. This mimics smear in many places, but only those that the plugin targets and not random locations throughout physical memory.

Line 3 of the algorithm simply iterates through each mutation currently implemented in the framework. Mutations are simple functions that are passed a buffer read from the memory sample and which return the buffer in a mutated state. Gaslight currently uses seven categories of mutations, as described in the next section. On Line 4 and Line 5, the wait queue is filled with all the parameters needed for the fuzzing component to properly test the memory sample. By running the fuzzer through every variation of *fuzz_past*, *read_number*, and implemented mutation, the fuzzing harness successfully and fully mutates each read operation performed by the memory forensics framework being tested.

Currently implemented mutations

The following list describes the mutations currently implemented in our fuzzing framework:

1. Fill the given buffer with all zero (0x00) values.
2. Fill the given buffer with all hex 0xff values.
3. Fill the given buffer with all randomly generated values.
4. On each boundary of 2, 4, 8, and 128 bytes, fill the boundary byte with 0x00, 0xff, or a randomly generated byte.
5. On each boundary of 2, 4, 8, and 128 bytes, fill the entire boundary (2, 4, 8, or 128 bytes) with 0x00, 0xff, or randomly generated bytes.
6. On each boundary of 2, 4, 8 and 128 bytes, fill the boundary byte with the current boundary value plus or minus 2, 4, 8, 128, or 4096.
7. On each boundary of 2, 4, 8 and 128 bytes, fill the entire boundary with the current byte value plus or minus 2, 4, 8, 128, or 4096.

The mutation sets described in 1, 2, and 3 were chosen as they closely mimic the effects of smear inside memory captures.

Generating buffers of random values simulates what memory forensics tools must process when smearing of page table entries results in a page being replaced with one corresponding to a different application or operating system memory region during acquisition.

Mutation sets 4 and 5 mimic smear that occurs when data structures are partially overwritten. Instead of overwriting the entire buffer, only select portions are. This occurs frequently in memory samples as memory forensic tools will either find the beginning of data structures through carving memory or by following pointers whose dereferenced data is now stale and partially overwritten. Handling and detecting this condition in memory forensics frameworks requires careful programming as some members of a data structure might be valid while others might not be.

Mutation sets 6 and 7 mimic a very difficult situation in memory forensics where a smeared pointer refers to a valid memory address, but not to the correct data. This is hard for memory forensics tools to detect as the referenced address will still pass physical address translation, but the data stored at the address is not actually associated with the data structure being processed.

Applying mutations

Once the work queue of fuzzing states is filled, the fuzzing harness then runs the configured memory forensics framework against each list of configuration options in the queue. The queue is processed by a Python worker pool, where each worker runs in a separate process. One worker is generated per available CPU core.

Each configuration operation per queue item is handled in the following manner:

- *sample* – The path to the memory sample, relative to the FUSE mount point.
- *profile* – The profile that describes the data structures for the sample. This is the convention used by both Volatility and Rekall.
- *plugin* – Which plugin to run. This can also include any plugin-specific options.

These three options are passed to the configured memory forensics framework. The following values are passed using environment variables that are handled by the custom FUSE implementation that performs the fuzzing. The use of environment variables allows the fuzzing harness to pass information to the FUSE handler without modifying the memory forensic framework being tested:

- *fuzz_past*, as FUZZER_FUZZ_PAST
- *read_number*, as FUZZER_WHICH_TIME
- *mutation_index*, as FUZZER_WHICH_MUTATION

As the memory forensics framework parses the memory sample exposed by the FUSE filesystem, the custom filesystem's *read* operation handler consults the environment variables to perform the proper mutations. This can be seen in Gaslight's FUSE *open* handler, which is called when the memory forensics framework opens a handle to the sample:

```
def open(self, path, flags):
    full_path = self._full_path(path)
    fd = os.open(full_path, flags)
    fds[fd] = fuzz_tracking()
    fds[fd].which_time =
        int(os.environ['FUZZER_WHICH_TIME'])
    fds[fd].fuzz_past =
        int(os.environ['FUZZER_FUZZ_PAST'])
    mutation_index =
        int(os.environ['FUZZER_WHICH_MUTATION'])
    fds[fd].mutation =
        mutations.mutations[mutation_index]
    return fd
```

As this code snip illustrates, when the *open()* system call is invoked, the file descriptor that's returned is associated with the environment variables passed by the fuzzing harness. When the memory analysis framework subsequently reads data from the sample on the opened file descriptor, the following code is executed:

```
def read(self, path, length, offset, fh):
    os.lseek(fh, offset, os.SEEK_SET)
    buf = os.read(fh, length)
    if fds[fh].read_counter ==
        fds[fh].which_time:
        do_fuzz = True
    elif fds[fh].read_counter >
        fds[fh].which_time:
        do_fuzz = fds[fh].fuzz_past == 1
    else:
        do_fuzz = False
    if do_fuzz:
        buf = fds[fh].mutation(buf)
    fds[fh].read_counter =
        fds[fh].read_counter + 1
    return buf
```

The read handler tracks the number of reads on the file descriptor and applies mutations, depending on whether the read counter has reached the value of the *which_time* variable and/or if *fuzz_past* is set once the counter is passed.

Automated detection of errors

As plugins are run through the fuzzer, their output, including data written to both the standard error and standard output file descriptors, is written to a unique file in an output directory. This file is named using the following convention:

<plugin-name>-<read number>-<mutation index>-<fuzz past value>

This naming scheme is unique to all plugin runs and ensures that output from one run does not overwrite the output of a previous run. Furthermore, the name encapsulates the exact state of the fuzzer when it caused the tested memory forensics framework to produce a processing error. Being able to recreate this state on demand substantially reduces the effort needed to triage processing errors.

To automatically detect errors generated by the fuzzer, the following steps are taken:

1. When the fuzzing harness executes the memory forensics framework, it implements a timeout value that triggers if a plugin has taken too long to run. The default value for this timeout is 5 min. We are aware, of course, that many memory scanning plugins can take much longer to run. Fuzz testing these plugins currently requires disabling the timer or significantly increasing its value.
2. The output files of all the plugin runs are searched for a case-insensitive set of strings, e.g., “traceback”, “exception”, etc. This step is useful for Python frameworks that produce back traces when unhandled exceptions are triggered. It can trivially be modified for other frameworks that implement global exception handlers and custom output when exceptions are caught.
3. The size of the output files is checked. For plugins with text-based output, they are filtered for sizes over 1 KB. For file extraction plugins, the size check is 1 MB. The purpose of this check is to look for plugin errors that do not crash the program, but for which processing produces large, erroneous output. An example is a plugin that walks a mutated loop pointer and produces hundreds or thousands of lines of output instead of detecting and avoiding the bad pointer. File extraction plugins are checked to ensure that they validate metadata related to file sizes and offsets before extraction. Otherwise, file system exhaustion attacks are possible.

Fuzzing setup

Hardware

Our fuzzing efforts were performed on commodity hardware. Initial testing was performed using an Alienware 13 R3 laptop with 32 GB RAM and an Intel i7 processor with 4 cores (8 threads) and a desktop with 32 GB of RAM and an Intel i7 processor with 6 cores (12 threads). Performance of the current harness is related directly to the number of cores available on the system used for testing. The memory analysis plugins we tested all use less than 1 MB of RAM, so memory constraints were not a factor. We've deferred an extensive performance analysis until the fully distributed version of Gaslight is ready.

Memory samples

To illustrate the usefulness of our framework, we chose one memory sample per major operating system to test against. The following table lists the memory size and operating system version of each sample. The 32-bit Linux sample is presented simply for variety, not because of limitations in Gaslight.

Operating system version	Memory size
Windows 7 SP1 64-bit	2 GB
Debian Wheezy 32-bit	2 GB
Mac OS X Sierra 64-bit	4 GB

In the Future Work section we discuss our ongoing research to identify the variety of operating systems needed for full memory forensic testing coverage.

Memory forensics frameworks

As a proof of concept, we chose to test two widely used memory forensics frameworks, both of which are commonly utilized in real-world investigations.

The bulk of our effort focused on the Volatility framework, since it is widely used in the field and because we have extensive experience with it. To show wide applicability, we also tested Rekall. The results of testing each framework are described next.

Fuzzing results – volatility

The following sections list the plugins for which Gaslight triggered an anomalous or error condition within a particular Volatility plugin. We note that our tests were not exhaustive for each operating system in that we only chose 10–12 plugins per operating system to test. For this study, we chose frequently used plugins, while purposely avoiding plugins that would generate a very large number of read operations, such as those that scan all of physical memory or that scan the entire virtual address space of each active process. This was simply to maximize the number of plugins we could test in our research timeframe. This was not a result of any limitations of our fuzzing architecture or testing harness.

Results against linux plugins

The following plugins crashed because of mutations by Gaslight:

- *linux_library_list* – Crashed because of insufficient checks by the list enumeration code, leading to processing of invalid data.
- *linux_dmesg* – Crashed due to not checking if *log* structures were instantiated on a valid page before referencing members.
- *linux_arp* – Crashed because of no checks for integer overflow (Python's OverflowError) when performing a bit shifting operation to determine an array's size. The fuzzer mutated the buffer holding the number being shifted and caused the calculation to overflow.

The following plugins entered infinite loops due to mutations applied by Gaslight:

- *linux_bash* – Entered an infinite loop because the mutation applied by Gaslight coerced two list members to point back to each other and the list enumeration code did not check for this condition.

- *linux_arp* – Entered an infinite loop due to inadequate corruption checking by the list walking code, specifically, not checking if a pointer to be processed was already processed. This bug was found after fixing the previous bug in *linux_arp*.

Mutations introduced by Gaslight caused the following plugins to produce extraordinarily large amounts of data:

- *linux_psaux* – This plugin relies on non-critical members of the memory mapping data structure, *mm_struct*, which specifies where in userland memory the command line arguments start and stop. When the fuzzer mutated the buffer holding these values, the resulting size calculation caused the plugin to output gigabytes of erroneous data.
- *linux_psenv* – When attempting to process the starting and ending addresses of the process' environment variables, it produced large amounts of data due to the same issue as *linux_psaux*.

The following plugins attempted to generate extremely large files because of mutations applied by the fuzzer and no bounds checking on the size of the ELF files being produced.

- *linux_procdump* – This plugin extracts process executables to disk and generated enormous output files because a function it relied on, *write_elf_file*, did not perform adequate bounds checking.
- *linux_librarydump* – *write_elf_file* is also used by this plugin, to extract the process executable as well as loaded shared libraries, so it is impacted in the same way by lack of proper bounds checking.

Results against OS X plugins

The following plugins crashed because of mutations by Gaslight:

- *mac_check_syscall* – Crashed because the system call table array was instantiated without validating that it resided on a mapped page.

The following plugins entered infinite loops due to mutations applied by Gaslight:

- *mac_lsmod* – Entered an infinite loop because the plugin did not check for previously seen list members. We have seen this type of corruption happen frequently in memory samples from real investigations, and Volatility already checks for this condition in other list walking algorithms that it implements.
- *mac_lsof* – Entered what was essentially an infinite loop as the fuzzer mutated the structure member that specified how many file handles a process had opened. In our test case, the fuzzer mutated the buffer to be a value over 3 billion, which forced the plugin to attempt to loop and process memory that many times.

Mutations introduced by Gaslight caused the following plugins to produce extraordinarily large amounts of data:

- *mac_dyld_maps* – Entered an infinite loop and did not validate the library structure before printing it to the command line. This led to a file system resource exhaustion issue.
- *mac_psaux* – This plugin did not validate the number of arguments as specified in the memory map data structure. The fuzzer mutated this member to a very large integer, which caused the plugin to produce a huge amount of incorrect output.

Results against windows plugins

Our testing was not able to produce any crashes against the subset of Windows plugins we evaluated. We believe that this occurred because Volatility's Windows plugins are substantially more mature and have received more testing in the field than the Linux and Mac plugins. As a result, many Windows plugins are quite robust against corrupt data. An example which backs this argument is a recently closed bug in Volatility's *vaddump* plugin (Govers, 2016; Ligh, 2016). This bug was brought to the attention of the developers when a user reported that the *vaddump* plugin was attempting to produce huge (2 TB) output files. To remedy this issue, the developers set a maximum size limit of 1 GB for output files.

Since this patch was applied before we started testing Volatility with Gaslight, we obviously did not find it. To test the validity of our fuzzing architecture, we ran a custom instance of the fuzzer which only executed the *vaddump* plugin against a version of Volatility with the patch reverted. In this testing, the size issue with *vaddump* was triggered. This result illustrates that Gaslight is effective against frameworks that analyze Windows samples.

Summary of results

As discussed, Gaslight automatically found numerous issues in Volatility plugins that would lead to incomplete processing. The programming errors discovered by Gaslight include crashes resulting from a lack of exception handling, infinite loops due to insufficient memory corruption checks, and the generation of output files so large that they were unable to be reasonably examined. We are currently working with the developers to resolve these issues.

Fuzzing results – rekall

After finding numerous programming bugs in Volatility plugins, we then wanted to verify the flexibility of Gaslight to test other memory forensics frameworks. Due to time constraints, we were unable to test a large number of Rekall plugins. Instead, we chose to test its *arp* plugin, which targets Linux memory samples, since two crashes were found in the Volatility implementation. We do plan to perform comprehensive tests of both Volatility and Rekall in the future.

During our testing of Rekall's *arp* plugin, three issues were discovered. The first issue was the same as one from Volatility, in which a corrupted list will lead to infinite processing due to list members pointing to valid, non-repeating memory addresses. The second issue was similar to Volatility's issue in handling the bit shifting operation, but with an interesting twist.

Upon inspection of the source code of Volatility and Rekall, we determined that Volatility uses Python's *integer* type, which is limited to 32 bits. This 32 bit value is what overflowed when the mutated shift value was too large. Rekall, on the other hand, uses Python's *long* type, which can be extended to infinity. This led to Rekall running in essentially an infinite loop when the fuzzer mutated the shift result to grow to many trillions. Gaslight was able to automatically detect both the crash of Volatility as well as the long runtime of Rekall.

The last issue found was the insufficient validation of the data structures that track ARP entries. This issue led to a filesystem exhaustion issue, where a nearly infinite loop continuously wrote malformed data structures to disk. The Volatility plugin did not exhibit this behavior. Examination of the source code of the Volatility plugin shows that it validates several structure members, which eliminates this issue.

The ability for Gaslight to quickly be configured for and find bugs in different frameworks highlights its high degree of usability and applicability both now and in the future. The difference in handling of Python's integers and longs in Volatility versus Rekall also shows how Gaslight is able to find a variety of programming issues and to automatically detect a framework's anomalous behavior when encountering such discrepancies.

Conclusions and future work

In this paper, we have described a robust fuzzing architecture named Gaslight, which supports seamless testing of any memory forensics framework. To show the framework's efficacy, we demonstrated that Gaslight was able to find crashes in numerous core Volatility plugins. We also showed that Gaslight can be utilized against other memory analysis frameworks.

Gaslight is currently able to utilize all local cores in an efficient manner, so our next performance improvement is to automate scaling to as many systems as are available. To meet this goal, and to allow for testing of all Volatility and Rekall plugins against a wide range of memory samples, we are currently developing a cluster-based, distributed implementation of Gaslight. The implementation will run a large number of tasks in a distributed computing environment in parallel and include a task manager to coordinate creation and management of individual fuzzing tasks across the cluster. It will also include the scheduling of individual workers to check the output of plugins to determine crashes and suspiciously large amounts of output data. The reporting of such conditions will be automated, such as through email and/or a creation of a support ticket on GitLab. Our eventual goal is to have Gaslight running 24/7/365 against a wide variety of memory samples and utilizing a wide variety of frameworks.

We are also reviewing the source code bases of Volatility and Rekall to identify which kernel versions of Linux, Mac, and Windows made changes that broke existing algorithms implemented by memory forensic plugins. For example, Microsoft has frequently changed its implementation of how a process' memory regions (VADs) are tracked. Each of these implementation changes in Windows has required updates to memory forensic frameworks to support the new operating system version. Similar updates across operating systems has affected nearly every memory forensic plugin. For Gaslight to fully test the algorithms of a memory forensics plugin, it is necessary for its testbed of memory samples to include each operating system version where such changes occurred. Once this study is complete, we plan to generate memory samples for all the operating systems versions needed to fully test each plugin. Finally, after some code cleanup, Gaslight will be released as an open source project.

References

aim4r, 2015. VolDiff: Malware Memory Footprint Analysis Based on Volatility. https://github.com/aim4r/VolDiff.

Breen, K., 2015. Web App for Volatility Framework. https://github.com/kevthehermit/VolUtility.

Carvey, H., 2005. Page smear. http://seclists.org/incidents/2005/Jun/22.

Conti, J.J., Russo, A., 2010. A Taint Mode for Python via a Library. http://www.cse.chalmers.se/~russo/publications_files/owasp2010.pdf.

Conti, J.J., Russo, A., 2013. Writing a FUSE Filesystem in Python. https://www.stavros.io/posts/python-fuse-filesystem/.

Dolan-Gavitt, B., Srivastava, A., Traynor, P., Giffin, J., 2009. Robust signatures for kernel data structures. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, ACM, pp. 566–577.

Edward, D.B., Schwartz, J., Avgerinos, Thanassis, 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. In: Proceedings of the 2010 IEEE Conference on Privacy and Security, IEEE.

Foundation, T.V., 2016. The Volatility Framework: Volatile Memory Artifact Extraction Utility Framework. https://github.com/volatilityfoundation/volatility.

Gaynor, A., 2015. Introduction to Fuzzing in Python with AFL. https://alexgaynor.net/2015/apr/13/introduction-to-fuzzing-in-python-with-afl/.

Google, 2016. Announcing OSS-fuzz: Continuous Fuzzing for Open Source Software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

Google, 2016. Rekall. https://github.com/google/rekall.

Govers, R., 2016. Vaddump Issues with Win10x64_DD08DD42. https://github.com/volatilityfoundation/volatility/issues/333.

Habben, J., 2015. Web Interface for the Volatility Memory Forensics Framework. https://github.com/JamesHabben/evolve.

Ligh, M., 2016. Adding a 1GB Default Maximum Memory Range Size to Vaddump. https://github.com/volatilityfoundation/volatility/commit/8d10c1d81e4fbc4cdddcd6fcbb0e0ffcb52a61fb.

Marziale, L., 2014. Differential Analysis of Malware in Memory. https://github.com/504ensicsLabs/DAMM.

Miller, B., 1988. Fuzzing Creation Assignment. https://fuzzinginfo.files.wordpress.com/2012/05/cs736-projects-f1988.pdf.

Newsham, T., Palmer, C., Stamos, A., 2007. Breaking Forensics Software: Weaknesses in Critical Evidence Collection. https://www.defcon.org/images/defcon-15/dc15-presentations/Palmer_and_Stamos/Whitepaper/dc-15-palmer_stamos-WP.pdf.

Newsome, J., Song, D.X., 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the Network and Distributed System Security Symposium.

Prakash, A., Venkataramani, E., Yin, H., Lin, Z., 2015. On the trustworthiness of memory analysis—an empirical study from the perspective of binary execution. IEEE Trans. Dependable Secure Comput. 12 (2015), 557–570.

Zalewski, M., 2016. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

Zalewski, M., 2016. The Bug-o-Rama Trophy Case. http://lcamtuf.coredump.cx/afl/#bugs.