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.

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

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.