Detecting Memory Injections Using a Hardware Monitor

Marcus Botacin
botacin@tamu.edu
Texas A&M University
Texas, USA

Uriel Kosayev
uriel@cymdall.com
CYMDALL
Israel

Amichai Yifrach
amichal@cymdall.com
CYMDALL
Israel

ABSTRACT

Memory injection is the current state-of-the-art malware attack technique. Injections are hard to detect by current software-based AntiViruses (AVs) because monitoring operations system-wide causes significant performance impact. To mitigate performance penalties, AVs often only monitor specific parts of the system, thus naturally missing some injection points, that are actively exploited by the attackers in an arms race. A solution to the problem is to move AVs to hardware to allow full-system monitoring without performance impact. In this paper, we present a prototype of a hardware monitor to detect memory injection attacks. We evaluate the prototype via the injection of a backdoor payload into a native Windows process. The injection is not detected by the native Windows Defender, but it is detected by the proposed detector.

CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**; **Network security**; **Systems security**;

KEYWORDS

Malware, Antivirus, Cloud Computing, Hardware Design

ACM Reference Format:
Marcus Botacin, Uriel Kosayev, and Amichai Yifrach. 2023. Detecting Memory Injections Using a Hardware Monitor. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (ACSAC '23).* ACM, New York, NY, USA, 2 pages. https://doi.org/XXXXXXX.XXXXXXX

1 MEMORY INJECTION ATTACKS

In this work, we address the problem of memory injection by malware. We start presenting background on this type of threat to motivate the further presentation of a hardware-based detector.

Why Code Injection? OSes support adding code to processes for multiple reasons, from shrinking code bases via dynamic libraries to the addition of wrappers for legacy support. However, code injection might also become a threat when abused by malware.

The threat. Code injection becomes a threat when malware loads its code in the context of a third process. Malware does that for two reasons: (i) stealing information from the user by harvesting the process' data (e.g., browser information); and (ii) making detection harder, by mixing its malicious with the ones of a benign process.

How it works. There are multiple techniques to perform code injection [1]. They can be classified into two classes: (i) using native methods (e.g., Windows API); and (ii) abusing the OS structures. In the first case, the malware uses system APIs to load a pre-compiled code into a target process memory. In the latter, the attacker manually maps the payload into the image of the target process by manipulating the OS structures (e.g., the loaded image list).

How to Detect. When code injection is performed using the Windows APIs, one can directly monitor the API usage. The Windows kernel provides notification callbacks for this type of monitoring it. Typical AVs rely on these callbacks for monitoring code injection. When the injection strategy is based on the abuse of OS structures, the only detection method possible is to parse the OS structures and check their integrity. There is no API for that, which requires manual implementation. There is also no precise trigger for this check, which requires polling. Many AVs often do not implement this type of check for the above reasons.

2 DESIGN

We implemented two versions of the security monitor: (i) one in pure software, to serve as ground truth, and one in hardware, to be performance efficient. Both rely on the same detection engine. We following present the two implementations and describe the details of the detection engine.

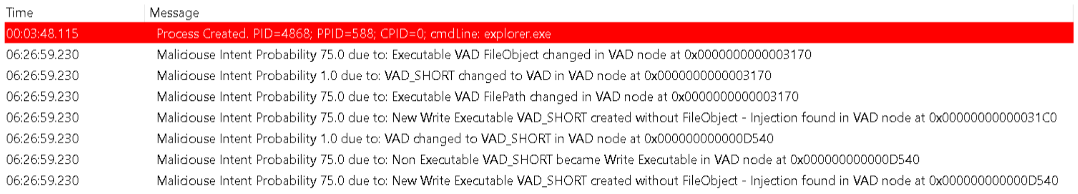
2.1 Software Implementation

In a software-based AV architecture, the AV has (i) a kernel component that accesses the memory objects at the OS memory subsystem; and (ii) a userland component that periodically requests kernel data to classify if the object is malicious or not. Figure 2 illustrates the data flow in this type of architecture as we implemented in our prototype. The major drawbacks of this architecture are that (i) the userland component has to synchronously query the kernel driver for data, which causes CPU impact; and (ii) the userland component has to spend CPU time processing the detection routines.

2.2 Hardware Implementation

In a hardware-based AV architecture, the AV does not need to have interposition components at the software level. Instead, the memory objects can be retrieved directly from the hardware structures–the kernel stores memory pages in the Memory Management Unit and the Translation Lookaside Buffer (TLB). Thus, the monitor implemented in hardware can access this information directly. Also, a hardware implementation does not require software for processing, since the detection engine can be implemented in the hardware layer. Figure 3 illustrates the data flow in this type of architecture as we implemented it in the hardware emulation solution. This architecture will be deployed in ASIC hardware.

ACSAC '23, June 03–05, 2018, Woodstock, NY
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX



```
Time            Message
00:03:48.115    Process Created. PID=4868; PPID=588; CPID=0; cmdLine: explorer.exe
06:26:59.230    Maliciouse Intent Probability 75.0 due to: Executable VAD FileObject changed in VAD node at 0x0000000000003170
06:26:59.230    Maliciouse Intent Probability 1.0 due to: VAD_SHORT changed to VAD in VAD node at 0x0000000000003170
06:26:59.230    Maliciouse Intent Probability 75.0 due to: Executable VAD FilePath changed in VAD node at 0x0000000000003170
06:26:59.230    Maliciouse Intent Probability 75.0 due to: New Write Executable VAD_SHORT created without FileObject - Injection found in VAD node at 0x00000000000031C0
06:26:59.230    Maliciouse Intent Probability 1.0 due to: VAD changed to VAD_SHORT in VAD node at 0x000000000000D540
06:26:59.230    Maliciouse Intent Probability 75.0 due to: Non Executable VAD_SHORT became Write Executable in VAD node at 0x000000000000D540
06:26:59.230    Maliciouse Intent Probability 75.0 due to: New Write Executable VAD_SHORT created without FileObject - Injection found in VAD node at 0x000000000000D540
```

Figure 1: Log Console. The hardware detector identifies the memory structure manipulation.

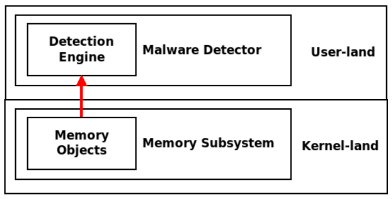


Figure 2: Software-Based Detector. In the traditional model, the information is collected from the kernel via a kernel driver and processed in the userland.

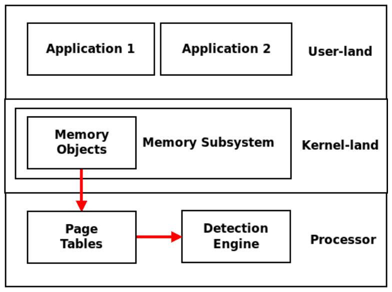


Figure 3: Hardware-Based Detector. In the new model, the kernel information is directly collected from and processed at the hardware level.

2.3 Detection Engine

The detection engine for memory injection is an integrity checker that enforces the security policy that only the libraries referenced in the PE header can be loaded in the memory space. To enforce that, we parse the PE header at the loading time and check all library names, creating a database for that process. When a new image is loaded within the process context, we check if that image matches the information in the database. If the information mismatches, code injection is detected and reported. In the software implementation, the loading of PEs and libraries is directly retrieved from the kernel via callbacks. In the hardware implementation, this information is retrieved from the loading of new memory pages. The parsing mechanism is implemented by directly checking the memory offsets within each read page.

3 EVALUATION

We here present the evaluation of the proposed solution against a real attack and real defenses.

The Attack. The sample attack for our evaluation consisted of a remote thread injection via the Asynchronous Procedure Call (APC) injection method [3]. This type of injection is often performed by malware samples in post-exploitation steps. The payload is injected into the native Windows' `explorer.exe` process. The injected payload consists of a reverse shell generated via Metasploit [2].

The Defenses. Our experiment considered two different scenarios: (i) with and (ii) without defenses. The scenario without defenses was only considered as ground truth to warrant the attack working. The scenarios with defenses considered four different protections, divided into three classes: (i) the developed hardware monitor; (ii) the native Windows defender; and (iii) two popular Endpoint Detection and Response (EDR) solutions.

Results. Once the attack succeeds in the ground truth experiment, we evaluate its effectiveness against the proposed defenses. Table 1 details the performance result of the defensive solutions.

Table 1: Detection Results. The hardware detector is the only solution that detects the memory injection.

Solution	Hardware	Defender	EDR1	EDR2
Detection	✓	✗	✗	✗

We notice that the attack succeeded in all scenarios with software AVs, either they being the native Windows Defender or the commercial EDRs. The proposed hardware detector was the only solution able to detect the thread injection.

Figure 1 illustrates the console messages outputted by the hardware solution. We notice that the solution was able to keep track of the memory changes during all attack steps.

4 CONCLUSION

In this paper, we investigated the problem of efficiently detecting code injection attacks by malware samples.

Contributions. We presented a prototype of a hardware-assisted malware detector that can detect memory injection by externally parsing Windows structures. The detection was possible even in the case when the native Windows Defender did not detect the malicious payload. We expect these results might foster new hardware-assisted security developments.

Limitations. The current prototype implementation is limited to memory injection attacks and it is still not a complete replacement for software AVs, that also detect other types of threats.

Future Work. We will extend the hardware monitor to cover other types of attacks, such that it might replace the software implementation of an AV engine. Our first step was to partner with a company to embed the solution into a chip.

Acknowledgments. The authors thank CYMDALL for supporting the development of this research project.

REFERENCES

[1] Ashkan Hosseini. 2017. Ten process injection techniques: A technical survey of common and trending process injection techniques. *Endpoint Security Blog* (2017).
[2] Metasploit. 2020. How to use a reverse shell in Metasploit. https://docs.metasploit.com/docs/using-metasploit/basics/how-to-use-a-reverse-shell-in-metasploit.html.
[3] MITRE. 2020. Process Injection: Asynchronous Procedure Call.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009