Automated Post-Attack Analysis of Injected Payloads

Remco Vermeulen

Vrije Universiteit

Friday 28, 2011

Introduction

Figure: 12-Year Old Alex Miller Earns $3,000 From Mozilla for Finding a Critical Vulnerability

Current situation

- Buffer overflows remain prominent in today's software.
- Buffer overflows are actively abused to propagate malware.
- Attackers changed from attacking network services to attacking client applications.

How abused?

- Attackers abuse buffer overflows using remote code-injection attacks.
- Remote code-injection attack procedures:
  1. Inject code into a buffer of a vulnerable program.
  2. Redirect the execution to the code in the buffer.

Problem

- Available protection mechanism are unable to fully mitigate the threat.
- It is of importance to know the behavior of the injected payload.
- Currently, it is a specialized and time consuming manual process to obtain this information.
    - Payload is written in machine language.
    - Payload is intimately tied with the operating system.
    - Payload is often packed in one or more layers of protection.

Proposed solution

- Automatically analyze the injected payload by prolonging the execution in a contained environment to:
  - Reveal the behavior (i.e., used API functions)
  - Reveal implementation details (i.e., execution trace + runtime information, use of packers.)
- Target modern client attacks as a client-side honeypot.

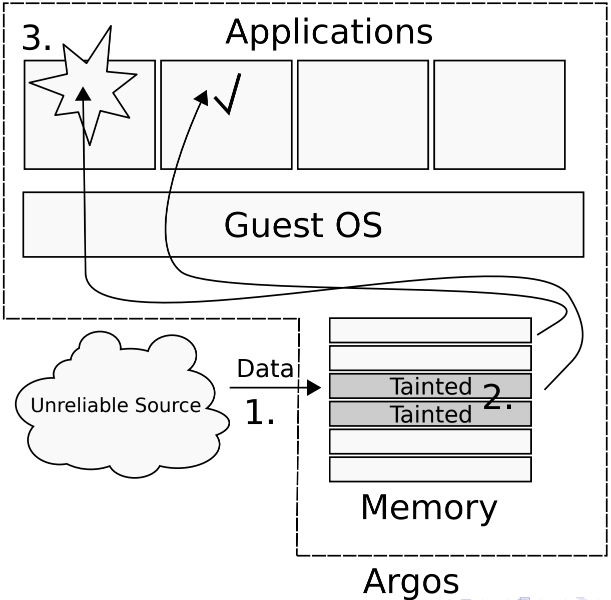
Procedures

- Use dynamic analysis to analyze the payload.
- To analyze the injected payload with dynamic analysis we have to:
    - Detect remote code-injection attacks.
    - Contain the executing payload.
    - Collect relevant information.

Attack detection

- Dynamic taint analysis detect remote code-injection attacks with high accuracy.
- Dynamic taint analysis currently is unable to replace existing protection mechanisms because of incurred overhead.
- Reused the dynamic taint analysis implementation of Argos, an ‘advertised’ honeypot.

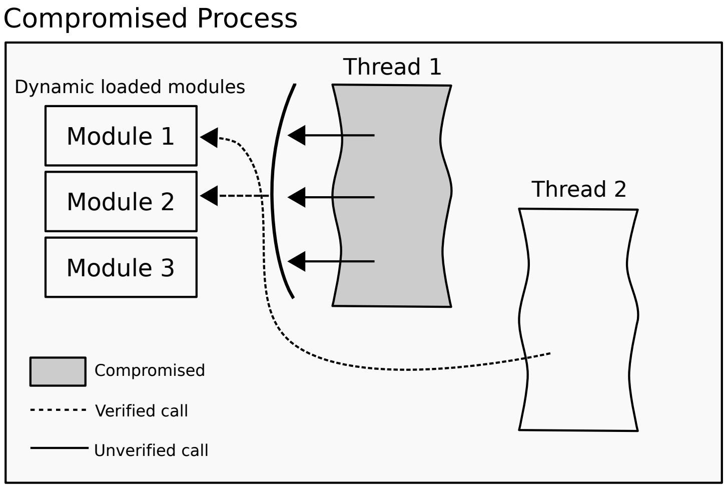
Dynamic taint analysis overview



Payload containment

- Executing malicious code means we can be used to attack other systems.
- Payloads, like any application, express behavior using API functions.
- To contain the payload we regulate the API usage of the payload using a whitelist

Payload containment procedure



Payload containment procedure

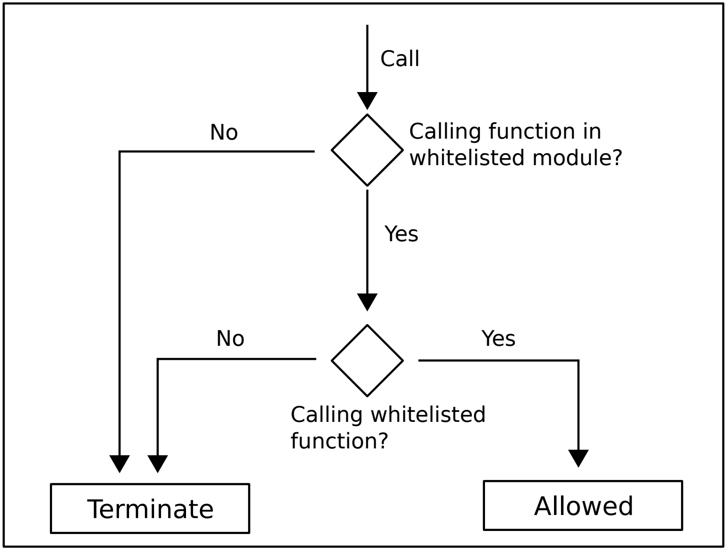
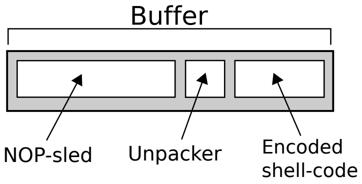

Call
Calling function in whitelisted module?
No
Yes
No
Yes
Calling whitelisted function?
Terminate
Allowed

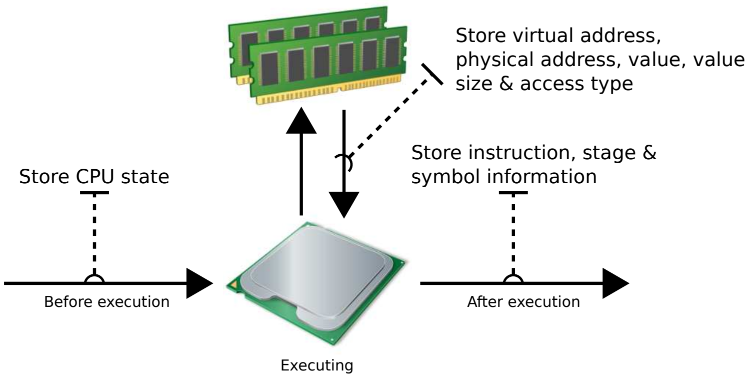
Figure: Payload identified with the CR3 register and the thread id.

Collecting of information

- High-level information for a overview of the behavior
  - Names of called API functions.
- Low-level information to see how the payload operates
  - Low-level instructions
  - CPU states
  - References to memory
  - Detect unpacking stages.



Details collecting of information


Store virtual address, physical address, value, value size & access type
Store instruction, stage & symbol information
Store CPU state
Before execution
After execution
Executing

Obtaining API function symbols

- API functions are made available by dynamic loadable libraries.
- The dynamic loadable libraries export API functions by name.
- We match these names to the addresses called by the payload.

Detecting unpacking stages.

- Unpacking of shell-code requires unpacked bytes to be written to memory.
- Each byte has stage zero.
- We update the stage on each write performed by the payload.
- The stage of the instruction is equal to the max stage of its bytes.

Evaluation

- We evaluated our solution in a real-world deployment as a client-side honeypot that visited known malicious domains.
- Compared the results with Shelia, a client-side honeypot that analyzes attacks on client applications.
- We presented an example case study that compares different payloads to find shared code using collected information (see thesis).

Evaluation results

- Both Argos and Shelia visited 57 domains flagged to contain browser exploits.
- Most domains were not available any more!
- Argos detected 5 attacks, logged 4 completed payloads and 1 partial because of an incomplete white-list.
- Shelia detected 4 attacks, logged malicious behavior of 3 payloads because on payload used anti-analysis techniques.

Future work

- Include analysis of payloads that use return-oriented-programming.
- Extend expressibility containment policy.
- Use static analysis to analyze parts of the payload that are not executed.

Conclusion

- With our real-world deployment as a client-side honeypot we were able to detect several remote code-injection attacks including one that uses anti-analysis techniques.
- None of the payloads were able to breach containment.
- With the collected information we were able to captured the behavior.
- With the collected information we were able captured implementation details.

Questions

Questions?