Using Memory Errors to Attack a Virtual Machine

Sudhakar Govindavajhala
Andrew W. Appel

IEEE Security and Privacy
2003
Executive Summary
Executive Summary

- Observation
- Key Idea and Implementation
- Key Results
- Takeaways
Executive Summary

- **Observation**
  - Type-checking systems are **safe** under the assumption that the computer **faithfully executes** its specified instructions.
  - This premise is false in the presence of **hardware faults**.

- **Key Idea and Implementation**
  - Write a program that uses memory errors to **overtake the system** by conducting a **type confusion attack**.
  - Manipulate **data placement** to maximize the probability of a memory error resulting in a type confusion.

- **Key Results**
  - Single bit errors in the program’s **data space** can be exploited to **execute arbitrary code** with a probability of ~70%.

- **Takeaways**
  - Virtual machines that employ type checking can be vulnerable to attacks that **exploit memory errors**
Outline

- Background
- Exploit
- Security Analysis
- Evaluation of the Attack
- Potential Countermeasures
Outline

- **Background**
  - Isolation
  - Type checking
  - Java applets and Java Cards
  - Memory Errors

- Exploit

- Security Analysis

- Evaluation of the Attack

- Potential Countermeasures
Background
Isolation

- Separate **trusted components** from **untrusted** ones
  - Example: **Virtual memory**
  - Each process operates in its **own virtual address space**

- **Type-checking** for **sound type systems**
  - Employed in virtual machines
  - Sound type system:
    - Rejects all **incorrect programs**
    - Every evaluation of an expression is **guaranteed to match** the expression’s static type
Type Checking

- What is type checking?
  - Verifying and enforcing constraints of types
  - Static vs. dynamic type checking

- Ensure type-safety
  - Do not allow operations/conversions that violate the rules of the system

- Why type checking?
  - Allows closer coupling between trusted and untrusted components
  - Object-oriented shared memory interfaces
  - No need for message passing / remote procedure calls
  - Same address space for trusted and untrusted programs
Type Checking in JVM

- At **compile time (static)**
  - Simulates program execution to determine if types are correct
  - After code is verified, it is trusted
  - Done by the bytecode verifier

- At **runtime (dynamic)**
  - No checks for type safety
  - Exceptions:
    - Casts
    - Array stores

- **Key assumption**
  - Read value is the same as when it was written
  - Time-of-check-to-time-of-use – the program changes after it was checked but before it was executed
Java Applets & Java Cards

**Applets**
- Program with few privileges
  - No *network* access
  - No access to the *file system*
- Executed in the JVM
  - Treated as *untrusted*

**Java Cards**
- Smart Cards
- Allow execution of Java Applets
- Store *secret information* (e.g. cryptographic keys, PIN)
Memory Errors

What are memory errors?
- Incorrect recall or complete loss of information in the memory system

Soft memory errors
- Single event upsets (SEU) – change of state in a single bit
- Transient – only lasts a short time
- Caused by some kind of disturbance (e.g., RowHammer)

Hard memory errors
- Permanent
- Error in the circuit (e.g. process defect)

Frequency of memory errors
- Once in several months (2003)
- About 10 a day per DIMM per Year

¹Bianca Schroeder et al., DRAM Errors in the Wild: A Large-Scale Field Study. SIGMETRICS 2009.
Causes of Memory Errors

- **Alpha particles**
  - Don’t penetrate matter well

- **Beta rays**
  - Interact too strong with plastic and metal packaging

- **X-rays**
  - Not enough energy
  - Not very portable

- **High-energy protons and neutrons**
  - Need a particle accelerator

- **Infrared**
  - Electronic components become unreliable at high temperatures
Related Research

- **D. Boneh et al., On the Importance of Checking Cryptographic Protocols for Faults. EUROCRYPT 1997.**
  - Used random hardware faults to recover secrets in cryptographic protocols.

  - Studies attack techniques on smartcards and other security processors by inducing errors at specific locations at specific points in time.
Outline

- Background
- **Exploit**
  - Threat Model
  - Type Confusion Attack
  - Attack Program
  - System Level Integration
- Security Analysis
- Evaluation of the Attack
- Potential Countermeasures
Exploit
Threat Model

- Target is a virtual machine that uses type checking as its protection mechanism

- Ability to provide a verified (type checked) program, which is loaded into memory and executed

- Physical Access to the machine

- No control over data memory of the program
Type Confusion Attack

- Circumvent the type-safety
  - Obtain references of different type that point to the same object

- Read or write to arbitrary location in the programs address space
  - Allows execution of arbitrary code
Attack Program

- Definition of two object types
  - Object size has to be a power of two
  - Assuming a 32-bit machine
- A acts as the pointer object
- B acts as the filler object

```java
class A {
    A a1;
    A a2;
    B b;
    A a4;
    A a5;
    int i;
    A a7;
}:

class B {
    A a1;
    A a2;
    A a3;
    A a4;
    A a5;
    A a6;
    A a7;
};
```
Memory Layout

- Allocate **one** object of type A
  - b field points to an arbitrary object of type B
- Allocate **as many** objects as possible of type B
  - All fields a1 to a7 point to the single object of type A

```java
class A {
    int a1;
    int a2;
    int b;
    A a4;
    A a5;
    int i;
    A a7;
};

class B {
    A a1;
    A a2;
    A a3;
    A a4;
    A a5;
    int a6;
    A a7;
};
```
Detecting a Bit Flip

- Wait for a bit flip to happen

- Detection of a bit flip
  - Iterate over all allocated objects of type B
  - Check if all references still points to the object of type A
  - Repeat until this is not the case anymore

- Assume the object of type A is at address $\mathbf{x}$ in memory
  - All references in objects of type B store the address $\mathbf{x}$
  - If a bit flip happens that reference stores an address that differs from $\mathbf{x}$
Implications of a Bit Flip

- Bitflip in bits 10 to 27
  - Reference address changes by more than the object size
  - Reference now points to header of B object
Implications of a Bit Flip

- Bits 2 – 9
  - Reference address changes by less than the object size
  - Reference now points within A object or adjacent object
Implications of a Bit Flip

- Other bits
  - Program crashes

- Very high order bits
  - Addresses point out of bounds (outside of the allocated heap)

- Very low order bits
  - Addresses are not properly aligned
Implications of a Bit Flip

Accessing the A object

\[
\begin{array}{c}
\cdots \\
\text{A header} \\
\text{A a1} \\
\text{A a2} \\
\text{B b} \\
\text{A a4} \\
\text{A a5} \\
\text{int i} \\
\text{A a7} \\
\cdots 
\end{array}
\]

x

Accessing the b field

\[
\begin{array}{c}
\cdots \\
\text{A header} \\
\text{A a1} \\
\text{A a2} \\
\text{B b} \\
\text{A a4} \\
\text{A a5} \\
\text{int i} \\
\text{A a7} \\
\cdots 
\end{array}
\]

x + offset
Implications of a Bit Flip

b384

B header
A a1
A a2
A a3
A a4
A a5
A a6
A a7

B header
A
A
A
A
A
A
A

B header
A
A
A
A
A
A
A

B header
A
A
A
A
A
A
A

B header
A
A
A
A
A
A
A

B header
A
A
A
A
A
A
A

B header
A
A
A
A
A
A
A
Implications of a Bit Flip

- New address + offset almost always stores the address \( x \)

```
A r;       B b384;       B q;
r = b384.a6;
q = r.b;
```

- \( q \) stores an address that points to the A object but has type B

- Now some reference \( p \) for the A object and \( q \) both store the address \( x \) but are of different type.
  \( \rightarrow \) Achieved a type confusion
Violating Type Safety

- Assume we have two references of different type that point to the same object.
- Type A reference \( p \) & Type B reference \( q \)
- Write address into integer field
- Interpret integer as an address

```c
A p;
B q;
int offset = 6 * 4;
void write(int address, int value) {
    p.i = address - offset;
    q.a6.i = value;
}
```
System Level Integration

- This allows **reading and writing of arbitrary addresses** in the address space of the **trusted process**

- Fill array with **machine code** and overwrite **virtual method table** with address of array

- Overwrite the **Security Manager**
  - Class that **enforces security policies**
Outline

- Background
- Exploit
- Security Analysis
- Evaluation of the Attack
- Potential Countermeasures
Security Analysis
Security Analysis

- Calculate **probability** of a single bit flip being exploitable

- Counting of “cousin” objects
  - Objects whose addresses differ by a single bit

- **Multiple bit flips** can be exploited with a **lower probability**
  - 6-bit error about **one-fourth** as likely to be exploitable
Outline

- Background
- Exploit
- Security Analysis
- Evaluation of the Attack
  - Methodology
  - Results
  - Exploiting before crashing
  - Safe bit flips
- Potential Countermeasures
Methodology
Methodology

- Two commercial JVMs from IBM and Sun on RedHat Linux
- Three different sets of experiments
  - Privileged Java thread inside that uses Interface to a C function that flips a bit in the processes address space
  - Unmodified JVM with separate Linux process that flips random bits in physical memory using /dev/mem
  - Unmodified JVM and induced memory errors by heating to 100 degrees Celsius
Results

### Attack Performance

![Bar chart showing attack performance for Process Memory, Physical Memory, and Heating for IBM's Java TM2 and SUN's Java TM2.](image)
Exploiting before Crashing & Safe Bit Flips

- Errors can **crash the system**
  - While **dereferencing** or **garbage collection**

- Probability of exploiting the error **before the system crashes** is about 71% according to measurements

- **Safe** bit flips
  - Only exploit bit flips in the bits 10 – 27
  - Bit flips in 2 – 9 are **indistinguishable** from flips in the extreme high/low order bits
  - This improves the exploit-before-crash ratio to 94%
Outline

- Background
- Exploit
- Security Analysis
- Evaluation of the Attack
- Potential Countermeasures
Potential Countermeasures
Countermeasures

- **Error correcting memory**
  - Use error correction codes to detect and correct errors
  - Memory overhead of 12.5% to detect 1-bit and 2-bit errors

- **Parity checking**
  - Parity bit stores parity of number of set bits

- **Software error logging**
  - Log occurring errors and adapt behavior
  - Disable untrusted software
  - Shut down

- Does not cover the whole datapath
Conclusion

- Observation
- Key Idea and Implementation
- Key Results
  - Single bit errors in the program’s data space can be exploited to execute arbitrary code with a probability of ~70%.
- Takeaways
  - Virtual machines that employ type checking can be vulnerable to attacks that exploit memory errors
  - Chosen program attacks alter the assumptions under which protection mechanisms should be designed
  - Hardware error-detection and correction with software logging of errors is the best defense
Critique
Strengths

- **Novel** idea
  - It was the first paper that used memory errors to take over a system

- **Relevance** to this day
  - Type confusion attacks are used to this day
  - Exploitation of memory errors escalated in relevance after the discovery RowHammer\(^1\) (2014)

- It inspired a lot of research

- **Strong** verification
  - They created a proof of concept

- Affects a high number of systems

---

\(^1\)Kim et al. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ISCA 2014
Weaknesses

- Very utopian threat model
  - Chosen program attack
  - Physical access

- No satisfactory protection mechanism

- Experimental results
  - Results only for one machine
  - Small sample size on heating experiment

- Writing is unstructured
Ideas & Takeaways

- Do **error correction in the processor** to solve the total datapath problem

- **Dynamic type checking** for dereferencing

- **Address Space Layout Randomization**\(^1\)
  - Randomly arranges the address space positions of key data of a process including base of the executable and the positions of the stack, heap and libraries.

- **Mark pages** as non executable/read only

\(^1\)PaX ASLR (Address Space Layout Randomization). 2003
Questions
Discussion
Is it possible to enable dynamic type checking with low performance overhead?
Related Research

  - Low-complexity architectural extension that replaces software-based dynamic type checking.
  - Automatic type checks for memory operations.

- *Dot et al. Removing checks in dynamically typed languages through efficient profiling. IEEE CGO 2017*
  - HW/SW hybrid mechanism that allows removal of checks in optimized code.
Are the current countermeasures insufficient and can you think of different protection mechanisms?
Ideas

- Do error correction in the processor to solve the total datapath problem

- Dynamic type checking for dereferencing

- Address Space Layout Randomization\(^1\)
  - Randomly arranges the address space positions of key data of a process including base of the executable and the positions of the stack, heap and libraries.

- Mark pages as non executable/read only

\(^1\)PaX ASLR (Address Space Layout Randomization). 2003
Can you think of any other attacks that could be performed in the same threat model?
Related Research

- **Halderman et al. Lest We Remember: Cold Boot Attacks on Encryption Keys. USENIX Security Symposium 2008.**
  - DRAM retains their content seconds to minutes after power is lost.
  - Perform a memory dump by cold booting a lightweight OS from a removable disk.
  - Circumvents full disk encryption.

- **Gruss et al. Rowhammer.js: A remote Software-Induced Fault Attack in JavaScript. DIMVA 2016.**
  - Fully automated attack to trigger faults on remote hardware.
  - Allows to trigger Rowhammer in highly restricted and even scripting environments by defeating complex cache replacement policies.
Should attacks like this be handled on a physical security level?
Are there other ways to exploit memory errors?
Related Research

- **Google Project Zero. Exploiting the DRAM rowhammer bug to gain kernel privileges. 2015.**
  - Achieving read-write access to one of its own page tables, and hence to all of physical memory.

- **V. van der Veen et al. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. ACM SIGSAC 2016.**
  - Shows that deterministic Rowhammer attacks are feasible on common mobile platforms.
  - Allows attackers to take control over the mobile device by hiding it in a malicious app that requires no permission.