

# Advanced Security

13

## Dynamic Symbolic Execution with Angr

DSE is a hybrid of concrete execution and symbolic execution

concrete exec = runs the binary with specific inputs and observes the behavior

symbolic exec = represents inputs symbolically and reasons about all possible behavior of the program

Angr uses DSE to identify paths in a binary that could lead to vulnerabilities.

- path exploration : Angr explores execution paths in a program. It uses symbolic execution to identify conditions under which specific paths are taken.
- constraint solving : it translates symbolic constraints into solvable equations to determine concrete input values that trigger vulnerabilities.
- exploitability analysis : can detect exploitable conditions, such as:
  - \* buffer overflows
  - \* use-after-free vulnerabilities
  - \* arbitrary memory reads or writes.

exercice 15: exploiting a buffer overflow to get an arbitrary read.

→ l'objectif : trouver une entrée qui fait un buffer overflow permettant à l'utilisateur de rediriger le pointeur "... " vers l'adresse du message "Good job" en mémoire, ensuite l'exécution binaire doit print "Good Job".

En faisant une exécution symbolique, il voit que quand l'entrée dépasse 16 bytes (taille de buffer), on peut écraser les variables voisines.

En générant des contraintes symboliques, Angr peut analyser comment locals\_to\_print peut être modifié pour pointer vers "Good Job".

Angr identifie les appels à la fonction puts, + précisément quand locals\_to\_print est contrôlable par l'entrée utilisateur.

Une fois que la contrainte est identifiée, Angr génère des contraintes symboliques pour vérifier que locals\_to\_print pointe vers l'adresse exacte de "Good Job".

Angr résout ces contraintes pour déterminer l'entrée exacte (valeurs de key et locals\_buffer) qui provoque cette état.

## Fault injection

## Fault injection attacks

Problem: faults targeting control signals

MAFIA: protection of the microarchitecture against fault injection attacks

CACPI: Code Authenticity and Control Flow Integrity

CSI: Control Signal Integrity

insertion of dummy instruction -O5 ← Read more on compiler

Random Precharging

"volatile" → évite que le compilateur supprime ce que fait la variable  
"la variable a pu être modifiée ailleurs"

Compiling with -O0

→ all program variables are moved onto the stack before anything else

→ bigger code size → larger attack surface ← more potential vulnerabilities.

Fault model: instruction skip

↳ cmp \$0, #1 msp

Can be protected by the duplication of idempotent instructions

Ex with Verify PIN()

-O0 ← pas d'optimisation et va donc générer un code binaire + gros = + de surface d'attaque

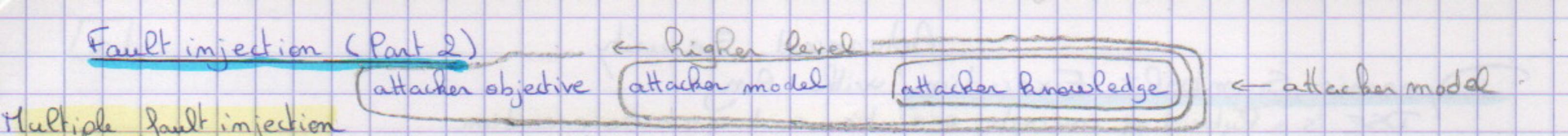
-O5 ← capable de faire sauter des instructions ← ici une comparaison

-O3 → target → runtime

Double la boucle for

→ faire attention avec quoi on compile

if we want to be protected against M+M faults → produce M+1 duplicates

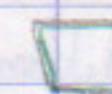
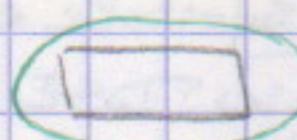
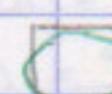


Multiple fault injection

Low Level Virtual Machine (LLVM)

Code analysis approaches

panel query	Under approximation methods
software	Over approximation method
	Hybrid method
	Exact method



Redundancy : les m<sup>es</sup> fautes mais peuvent <sup>être</sup> dans un ordre diff<sup>é</sup>rent

Equivalence : les m<sup>es</sup> fautes dans le m<sup>ême</sup> ordre

Lazart (the tool used here)

→ use `llc -make-symbolic` to have symbolic inputs

fault models: Test/Branch inversion

Data mutation

Jump

Switch call

→ cover most of high level fault models.

Countermeasure , 3 classes:

- detective countermeasure ← check some predicate during the execution
- infective countermeasure ← to infect the result      unusable  
program continue to run but the result will be →
- complete program modification ← ex: crypto /on change diversion

• Test Duplication

→ generates 2 detectors for each conditional branches

• SecSwift Control-Flow ← asserts that are detector

→ use XOR → kind of detector

• LBH's countermeasure ← protect against instruction skip and jump

→ insert step-counters for each C construct (checking macros are detectors)

Chaque contre mesure est contre un certain model

Multi fault make difficult to analyze programs (longer analysis times)

Faults models depends on the representation level (physical, architectural, binary, software, logical...)

Lazart uses DSE at LLVM level to produce multiple fault attacks tests cases.

## Code (de)Obfuscation

Tigress

Data obfuscation

Preventive Transformations

Control Obfuscation

Layout obfuscation

Source-level obfuscation against manual RE (Reverse Engineering)

↳ Stunmix obfuscator

- targets at obfuscating only the layout of the JavaScript code
- as the obfuscator parses the code, it removes spaces, comments and new line feeds
- while doing so, as it encounter user defined names → replaces them with random string
- replace print strings with their Hexadecimal values
- replaces integer value with complex equations.

Example of data obfuscation:

- Data re-encoding :
  - replace variables by complex expressions
  - replace standard arithmetic operations by more complex ones
  - obfuscate the data operations performed in the code.
- Data split : split some variables of type T<sub>1</sub>, into sets of variables of type T<sub>2</sub>
- Data merge : merge some variables of type T<sub>1</sub>, T<sub>2</sub> into a variables of type T
- Data fold or flatten arrays into higher / lower dimensional arrays.
- Convert static data into procedural data ("with a look-up table").

→ need alias computations and encoding / decoding functions

## Code (de) obfuscation (suite)

### Examples of code obfuscation

- **Opaque predicates**: control the control flow graph by inserting spurious conditions (evaluating always to true)
- **Virtualization**: turns a function into a interpreter by:
  - generating a bytecode instruction set
  - a bytecode array, a virtual program counter (VPC) and a virtual stack pointer (VSP)
  - a dispatch unit, and the bytecode instruction handlers.

### Anti-disassembling

#### Anti-dynamic analysis: prevent a program to be analyzed under a debugger, an emulator ...

- use process control primitives to prevent debugging (ptrace on Linux).
- try to access regular peripherals (network, printer, filesystem ...)
- monitor the execution time

### De-obfuscation techniques:

- **Static techniques** (de-assembling, de-compilation / static code analysis)
- **dynamic techniques** (debugging, code instrumentation, trace analysis, symbolic execution)

### Expected properties of an obfuscator

- **correctness**: should preserve the code semantics
- **resilience**: should prevent reverse engineering
- **cost**: should not "explode" the code complexity (time, memory ...)

However:

- no chance to build an universal obfuscator
- de-obfuscating tools are guided by existing obfuscation techniques... ↪ keep it secret

### 2 generally used techniques in static disassembly:

- **linear sweep** → drawbacks: if the text section contain data, the data will be treated as code and disassembled as instructions
- **recursive traversal**

### Obfuscation thwarts disassembly

- thwarting linear sweep: junk insertion
- thwarting recursive traversal: branch functions

### Obfuscation as it related to viruses

- Detect malware (Static & Dynamic)
- Virus types (Polymorphic, Metamorphic)

### Obfuscation techniques

## Race condition vulnerabilities

Happens when :- multiple processes access and manipulate the same data concurrently  
- the outcome of execution depends on a particular order.

When 2 concurrent threads of execution access a shared resource in a way that unintentionally produces different results depending on the timing of the threads or processes.

### Special type of race condition

- Time-of-check To Time-of-use (TOCTTOU)
- occurs when checking for a condition before using a resource.
- where the state of a resource changed between the time it is checked and the time it is used

Exploiting a race condition can lead to: privilege escalation / unauthorized access to resources / corruption or leakage of sensitive data.

- ex:
- ① checks if the file "/tmp/X" exists
  - ② if not, open system call is invoked
  - ③ there is a window between the check and use (opening the file)
  - ④ if the file already exists, the open() system call will not fail. It will open the file for writing.
- ⑤ So, we can use this window between the check and use and point the file to an existing file "etc/passwd" and eventually write into it.  
→ modifier un fichier admin avec des droits classiques.

### Countermeasures:

- atomic operations: to eliminate the window between check and use.
- repeating check and use: to make it difficult to win the "race"
- sticky SymLink protection: to prevent creating symbolic links.
- Principle of least privilege: to prevent the damages after the race is won by the attacker

## Reverse Engineering:

source level → model level

de-compiling: binary → source level

disassembling: binary → assembly level

- Main challenges:
- no symbolic information (meaningful function/variable/type names might be missing)
  - no explicit type information (only hint about the size and the sign of data accessed)
  - no structure information (functions, objects, classes ... are potentially lost)
  - no distinction between code and data
  - hard to modify/instrument

- Limitations:
- imprecision in decompiled code: require manual interpretation
  - limited context: reverse engineers have no access to the intent or design decisions behind code
  - encrypted/compressed binaries
  - dynamic dependencies: code relying on dynamic inputs, configuration files, or external network resources can be difficult to fully analyze statically.

### Static disassembling ← Ghidra

→ disassemble the whole file content without executing it.

main issue: distinguishing code vs data

### Dynamic disassembling ← GDB

→ disassemble the current instruction path during execution / emulation

main advantage: disassembling process guided by the execution.

- ensures that instructions only are disassembled
- the whole execution context is available (registers, flags, addresses, ...)
- dynamic jump destinations are resolved.
- dynamic libraries are handled

Blockchain

← des blocs et des chaînes

"Ethereum is an open blockchain platform that lets anyone build and use decentralized applications that run on blockchain technology".

① Alice écrit une transaction vers un smart contract enregistré dans la blockchain.

② La transaction déclenche l'exécution d'un smart contract présent dans la blockchain. La transaction à enregistrer est incluse dans le bloc en cours de formation.

③ Chaque "mineur" peut construire un bloc, respectant les règles communes, et le soumettre au consensus dans ses pairs.

⑥ Bob consulte les données enregistrées la blockchain en appelant une méthode du smart contract d'enregistrement.

⑤ Lorsqu'un accord (à la majorité) est obtenu, le bloc gagnant est confirmé. Il est ajouté à la blockchain,从而 compléter le ledger partagé et repliqué sur chaque pair du réseau.

④ Le bloc est vérifié par les pairs participants au consensus, au regard des règles communes. Lorsqu'un accord (à la majorité) est obtenu, le bloc gagnant est confirmé.

## 2 types de transactions

**Transaction de compte à compte:** ethereum account → change la balance de account source et account destination.  
↳ transférer des cryptomonnaies d'un compte à un autre (sur la blockchain).

**Transaction de compte à smart contract:** ethereum account → déclenche l'exécution d'un code spécifique de la blockchain.

↳ enlever le facteur humain

↳ ex: crowdfunding :  

- les participants envoient de l'argent au contrat
- si la somme totale atteint un certain seuil avant la date limite, les fonds sont transférés au créateur du projet
- sinon les fonds sont remboursés automatiquement aux participants.

**Token :** programmation d'un token avec un smart contract

**Currency :** création de crypto monnaie grâce à l'imitation = crypto monnaie

↳ propriétés :

- **Fongible** : une unité est équivalente à une autre
- **Divisible** : chaque unité peut être divisée en petites unités de valeur
- **Acceptable** : largement accepté comme moyen d'échange
- **Création** : un mécanisme d'incentive permet de créer les unités
- **Quantité limitée** : la quantité d'unités en circulation est limitée et/ou plafonnée
- **Portable** : les unités peuvent être échangées contre une autre currency
- **Durable** : les unités peuvent être utilisées sans être dégradées

**Usage** → transfert d'argent / réserve de valeur / unité de compte ... pour déployer des smart contracts

**Token :** représentation numérique d'une valeur

**Token standard :** ethereum Request for Comment (ERC)

**Token fongible :** standard ERC-20, optimisé ERC-233, sécuritaire ERC-777  
un token fongible est interchangeable avec un token du même type.

**Token non fongible :** standard ERC-721

présente des caractéristiques uniques, ne peut pas être échangé avec un autre.  
peut avoir des valeurs différentes les uns des autres.

**Utility Token :**

- peut représenter un droit d'accès
- permet d'accéder à un bien ou à un service fourni via une blockchain

**Security Token :** standard ERC-1400

- hérite des caractéristiques des token fongible ERC-20  
- représente la propriété d'un bien ou d'une valeur  
- est régulé quant à l'identité, la juridiction ou la valeur utilisée.

Ethereum Improvement Proposal (EIP) = implementation des ERC en langage Solidity.  
Fournit le prototype des fonctions standards du smart contract de token ERC20 qui implémente un livre de compte accessible aux utilisateurs depuis leur adresse de compte.

## Hardware Security (Software security)

CSRAM vulnerabilities : retention and aging  
→ data retention more than 15 minutes after the current is off

### Cache memories attacks

- The time taken to access data varies according to whether or not it is present in the processor's cache
- The presence or absence of data in the cache is therefore information that can be retrieved by a side-channel attack based on the measurement of the execution time of an access to this data.
- Recovery of calculation time (linked from data access time) is a problem for cryptography algorithms → recovery of the possible key for RSA, ECC or AES

### Flush + Reload:

besoin que le cache soit partagé

① Flush (vide une ligne spécifique du cache)  
② Attente (exécution) → attaquant laisse la victime exécuter son programme  
③ Reload (recharge et mesure de temps) → l'attaquant accède à la même adresse mémoire que celle vidée / mesure le temps d'accès à cette adresse mémoire / accès rapide : la victime a recharge les données en cache / accès lent : la donnée n'a pas été recharge

### Evict + Reload:

↳ même idée que flush + reload mais ici evict (remplace les données ciblées).

### Prime + Probe:

← ne nécessite pas de mémoire partagée

### Flush + Flush attack:

Rowhammer attack: ~~adversary~~ → adversee able

essayer de provoquer un bit flip en accédant aux adresses au dessus et dessous.

Mitigation: Static analysis / Monitoring with a performance counter / Memory fingerprint / Error correcting code

Transparent Huge Page → reduce the number of pages 4KB → 2MB

Coldboot: les données mettent plus longtemps à disparaître dans le froid  
+ la RAM est froide, + les données persistent longtemps.

Meltdown: utilise le cache pour extraire des données

Spectre: prédiction de branches et exécution spéculative.

Take aways: All the performance gains achieved over the last decades have an impact on security.  
Forget about trying to share things if you want security!

Every bug or fault is an opened gate for exploitation towards leakage of data

The only limit is hacker's imagination!