

Software security, secure programming

Conclusion

Master M2 Cybersecurity

Academic Year 2023 - 2024

Software vulnerabilities

A multi-level issue . . .

- ▶ weaknesses in the **specification**:
unprotected data-flows wrt confidentiality/integrity
lack of input validation/sanitization, etc.

Software vulnerabilities

A multi-level issue . . .

- ▶ weaknesses in the **specification**:
unprotected data-flows wrt confidentiality/integrity
lack of input validation/sanitization, etc.
- ▶ incorrectnesses in the **algorithms**:
spatial/temporal memory errors, race conditions, etc.

Software vulnerabilities

A multi-level issue . . .

- ▶ weaknesses in the **specification**:
unprotected data-flows wrt confidentiality/integrity
lack of input validation/sanitization, etc.
- ▶ incorrectnesses in the **algorithms**:
spatial/temporal memory errors, race conditions, etc.
- ▶ (language related) **programming** errors:
bad use of APIs, vulnerable built-in functions or patterns,
undefined behaviors, side-channels, etc.

Software vulnerabilities

A multi-level issue . . .

- ▶ weaknesses in the **specification**:
unprotected data-flows wrt confidentiality/integrity
lack of input validation/sanitization, etc.
- ▶ incorrectnesses in the **algorithms**:
spatial/temporal memory errors, race conditions, etc.
- ▶ (language related) **programming** errors:
bad use of APIs, vulnerable built-in functions or patterns,
undefined behaviors, side-channels, etc.
- ▶ **Hardware dependent** issues:
Spectre/Meltdown, Rowhammer, side-channels, etc.

Software vulnerabilities

A multi-level issue . . .

- ▶ weaknesses in the **specification**:
unprotected data-flows wrt confidentiality/integrity
lack of input validation/sanitization, etc.
 - ▶ incorrectnesses in the **algorithms**:
spatial/temporal memory errors, race conditions, etc.
 - ▶ (language related) **programming** errors:
bad use of APIs, vulnerable built-in functions or patterns,
undefined behaviors, side-channels, etc.
 - ▶ **Hardware dependent** issues:
Spectre/Meltdown, Rowhammer, side-channels, etc.
- ↔ still very pregnant (CVEs) and source of concrete attacks!

Software vulnerabilities

A multi-level issue . . .

- ▶ weaknesses in the **specification**:
unprotected data-flows wrt confidentiality/integrity
lack of input validation/sanitization, etc.
- ▶ incorrectnesses in the **algorithms**:
spatial/temporal memory errors, race conditions, etc.
- ▶ (language related) **programming** errors:
bad use of APIs, vulnerable built-in functions or patterns,
undefined behaviors, side-channels, etc.
- ▶ **Hardware dependent** issues:
Spectre/Meltdown, Rowhammer, side-channels, etc.

↔ still very pregnant (CVEs) and source of concrete attacks!

. . . enhanced by the (incorrect) use of insecure languages

- ▶ importance of **type safety** and **memory safety**
- ▶ trade-off between safety/security and run-time efficiency
(execution time, resource consumption)

↔ things may move slowly (JavaScript → TypeScript; C → Rust?)

Protections and mitigations

- ▶ A huge amount of available **secure coding** documentation (CWEs, “secure coding patterns”, books, etc.)

Protections and mitigations

- ▶ A huge amount of available **secure coding** documentation (CWEs, “secure coding patterns”, books, etc.)
- ▶ A wide-spectrum set of **protection mechanisms**
 - ▶ compilation options for code hardening
canaries, CFI, etc.
 - ▶ (lightweight) runtime error detection tools
adSan, UBSan, Valgrind, etc.
 - ▶ OS-level protections
DEP, ASLR, etc.
 - ▶ hardware mechanisms (TEE, memory enclaves)
ARM TrustZone, Intel SGX, etc.

Protections and mitigations

- ▶ A huge amount of available **secure coding** documentation (CWEs, “secure coding patterns”, books, etc.)
- ▶ A wide-spectrum set of **protection mechanisms**
 - ▶ compilation options for code hardening canaries, CFI, etc.
 - ▶ (lightweight) runtime error detection tools adSan, UBSan, Valgrind, etc.
 - ▶ OS-level protections DEP, ASLR, etc.
 - ▶ hardware mechanisms (TEE, memory enclaves) ARM TrustZone, Intel SGX, etc.

↪ widely deployed on main stream execution platforms . . .

but **take care with more specific ones** (IoT, Scada, etc.)!

Code analysis techniques & tools

Goals

- ▶ vulnerability detection
- ▶ vulnerability analysis (e.g., to evaluate their exploitability level)
- ▶ reverse-engineering and/or forensic analysis assistance
- ▶ code (de-)obfuscation, etc.

Code analysis techniques & tools

Goals

- ▶ vulnerability detection
- ▶ vulnerability analysis (e.g., to evaluate their exploitability level)
- ▶ reverse-engineering and/or forensic analysis assistance
- ▶ code (de-)obfuscation, etc.

Several approaches

Mostly adapted from safety-oriented code verification techniques

- ▶ static techniques: syntax-checking, pattern detection, static analysis
- ▶ dynamic techniques: fuzzing, (Dynamic) Symbolic Execution

Code analysis techniques & tools

Goals

- ▶ vulnerability detection
- ▶ vulnerability analysis (e.g., to evaluate their exploitability level)
- ▶ reverse-engineering and/or forensic analysis assistance
- ▶ code (de-)obfuscation, etc.

Several approaches

Mostly adapted from safety-oriented code verification techniques

- ▶ static techniques: syntax-checking, pattern detection, static analysis
- ▶ dynamic techniques: fuzzing, (Dynamic) Symbolic Execution

A strong **decidability** issue:

- ▶ no way to get a fully automated bullet-proof security insurance!
- ▶ trade-off between **soundness** (no false negatives) vs **completeness** (no false positives)

Code analysis techniques & tools

Goals

- ▶ vulnerability detection
- ▶ vulnerability analysis (e.g., to evaluate their exploitability level)
- ▶ reverse-engineering and/or forensic analysis assistance
- ▶ code (de-)obfuscation, etc.

Several approaches

Mostly adapted from safety-oriented code verification techniques

- ▶ static techniques: syntax-checking, pattern detection, static analysis
- ▶ dynamic techniques: fuzzing, (Dynamic) Symbolic Execution

A strong **decidability** issue:

- ▶ no way to get a fully automated bullet-proof security insurance!
- ▶ trade-off between **soundness** (no false negatives) vs **completeness** (no false positives)

And still a challenging issue to analyse **binary** code . . .

Outline

What have been seen?

Future trends?

A few words on CodeQL

A few words on machine-learning techniques

Software vulnerabilities

Probably **not over** in a near future . . .
(endless cat and mouse games between attackers & defenders)

but:

- ▶ “basic” vulnerabilities (BoF, arithmetic overflows, etc) should become less prominent . . .
- ▶ more HW/SW security issues?

Vulnerability **exploitation** should become more and more difficult on **recent** execution platforms . . .

but still a huge panel of **legacy/unprotected** hardware and software (e.g., in industrial systems)

Vulnerability detection & analysis tools

For the code developers

From DevOps to **DevSecOps**, with a potential increase of:

- ▶ fuzzing
- ▶ pattern-based detection tool (like CodeQL)
- ▶ machine-learning techniques (?)

For the code auditors & security experts

Towards (smarter) combinations of:

- ▶ fuzzing, dynamic-symbolic execution and static analysis
- ▶ machine-learning techniques ...

Possibly with an emphasis on **quantitative** analysis
(**how much** dangerous is a vulnerability?)

Outline

What have been seen?

Future trends?

A few words on CodeQL

A few words on machine-learning techniques

CodeQL: an example of pattern-detection tool

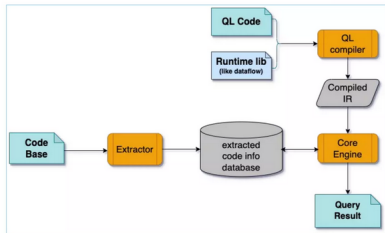
- ▶ static analysis (no code execution!)
- ▶ allows to find “arbitrary” **patterns** on (large) code bases
↳ e.g., to look for existing CWEs
- ▶ offer a powerful **query language** for pattern description allowing to mix syntactic and **semantic** features, including:
 - ▶ data-flow analysis
 - ▶ range analysis
 - ▶ alias analysis
 - ▶ etc.
- ▶ easy to integrate within a CD/CI pipeline ...

What is CodeQL

- Founded in 2006, a research project from Oxford University, acquired by Github in 2019
- Partly open source , but the core engine is source closed
- Basically scan code, make database and run query logic, find patterns
- Make code analysis to code property query

[Credits: H. Zhang, **CodeQL: also a powerful binary analysis engine** - BlackHat 2023]

Architecture of CodeQL



- The extractor can be regarded as language frontend, so it's language depends
- Extractor scan code, make extra analysis and store code property to database
- Database store code information, can be shared and reused
- CodeQL introduced a query language, the ql is related with query logic, but not related with code that being analyzed, so it's language agnostic
- CodeQL has developed a mature and comprehensive library that can perform various data flow analysis, such as the classic taint analysis
- The core engine can be regarded as a database evaluate engine

[Credits: H. Zhang, **CodeQL: also a powerful binary analysis engine** - BlackHat 2023]

Query examples and demo

Example of C/C++ publicly available queries

Demo of a use-after-free query ...

Outline

What have been seen?

Future trends?

A few words on CodeQL

A few words on machine-learning techniques

Examples of ML applications for cybersecurity

ML techniques

- ▶ **supervised ML:**

Use **labeled dataset** to train algorithms and define the variables to be assessed for correlations (input/outputs being specified). Model weights can be adjusted to avoid overfitting or underfitting.

- ▶ **reinforcement ML:**

Train the algorithm by **trial and error** rather than using sample data.

- ▶ **unsupervised ML** (used for deep-learning):

Analyze and cluster **unlabeled datasets** to identify hidden patterns or data clustering.

Application to Cybersecurity

- ▶ Intrusion detection (computer, network)
- ▶ Malware/ransomware detection & recognition
- ▶ Anomaly detection
- ▶ Log aggregation/corelation & alert analysis (e.g. in SIEM systems)
- ▶ **vulnerability** detection and analysis ...

ML for vulnerability analysis

Main challenges

- ▶ get a rather **balanced** sets of (labeled?) vulnerable & non vulnerable code examples
- ▶ relevant code features include **data-flow** and **control-flow** information, to be properly extracted & processed to feed the models

Main applications

- ▶ vulnerability **detection** (or simply “vulnerable code” detection . . .)
- ▶ reverse engineering: function detection, type identification, binary diffing, etc.
- ▶ enhanced code analysis techniques (fuzzing, pattern recognition)
- ▶ side-channel & information leakage detection
- ▶ etc.

examples of recent papers

A typical Vulnerability detection tool

VulDeePecker: A Deep Learning-Based System for Vulnerability Detection
(NDDS Conference, 2018)

So, what about ML for Software Security?

Not yet the “definite solution” for vulnerability detection/analysis:

- ▶ hard to evaluate and compare with other existing techniques
- ▶ lack of result **explainability**
(e.g., correctly locating vulnerable statements?)
- ▶ what about **new** vulnerability patterns?

But clearly a promising and essential research direction . . .

in conjunction with classical techniques

A possible next challenging (and more practical) step:

using **generative IA** to produce **secure-by-construction software**?

Credits

A Survey on Machine Learning Techniques for Cyber Security in the Last Decade - K. Shaukat et al- IEEE Access 2020

Machine learning (ML) in cybersec2yyurity - SailPoint

ML4Sec papers