INF226 – Software Security

Håkon Robbestad Gylterud

2019–09-02

STRIDE and SQL injection

- **Spoofing**: Transmissions with intentially mislabeled source.
- **Tampering**: Modification of persistent data or data in transport
- **Repudiation**: Denial of having performed unauthorized operations, in systems where these operations cannot be traced.
- **Information disclosure**: Access to data in an unauthorized fasion.
- **Denial of Service**: Rendering a service unaccessible to intended users.
- **Elevation of priviledge**: Non-priviledged users gaining access to priviledged operations and data.

Trusting trust

Trusting trust

Thompson's argument is a *reductio ad absurdum* of the statement:

> *It is sufficient to inspect the source code of a program to determine its behavior.*

Trusting a program from source

- To trust a program after reading the source code **we must trust the compiler to compile correctly**.
- To trust the compiler we can read the source code, but without trusting the compiler we cannot trust the resulting executable.

Conclusion: to trust the compiler we must trust the compiler, which is circular.

Compiler bootstrapping

Compiler bootstrapping

In the article, Thompson presents idealised code from a compiler:

```
c = next();
if(c != '\\')
  return c;
c = next();
if (c == '\\')
  return '\\';
if (c == 'n')
  return '\n';
```

Compiler bootstrapping

In the article, Thompson presents idealised code from a compiler:

```
c = next();
if(c != '\\')
  return c;
c = next();
if (c == '\\')
  return '\\';
if (c == 'n')
  return '\n';
```

Question: How can this code work, when the ASCII values it is supposed to produce (i.e. '\n' is 10), is not in the source?

The deceptive compiler (1st level)

A compiler could try to recognise that it is compiling the `login` command of the OS:

```
if(match("pattern of login")) {
   compile("backdoor");
}
```

. . . and then compile in *a back door*.

The deceptive compiler (2nd level)

To avoid detection by reading compiler source code: Recognise when you are compiling the compiler, and write in the `login` modification, in the same way.

```
if(match("pattern of login")) {
   compile("backdoor");
}
if(match("pattern of compiler")) {
   compile("login backdoor inserter");
}
```

Questions

1. Are interpreted languages (such as python) immune to this threat?
2. What other programs could have a similar (linchpin) rôle w.r.t. OS security?

Diverse double compiling

Functional equivalence

Two programs, X and Y, are **functionally equivalent** if the output of X is the same as the output of Y when they are given the same input.

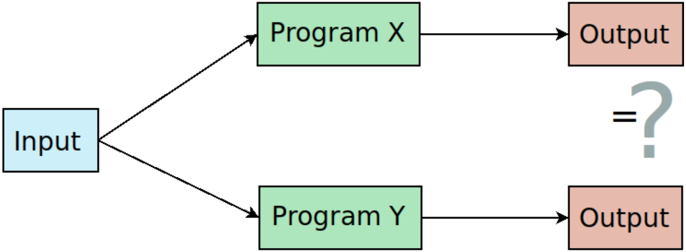

Figure 1: Functional equivalence

Functional equivalence

Two programs, X and Y, are **functionally equivalent** if the output of X is the same as the output of Y when they are given the same input.

Functional equivalence

Two programs, X and Y, are **functionally equivalent** if the output of X is the same as the output of Y when they are given the same input.

Question: Can an implementation of Bubble Sort be functionally equivalent to an implementation of Quck Sort?

Functional equivalence

Two programs, X and Y, are **functionally equivalent** if the output of X is the same as the output of Y when they are given the same input.

Question: Can an implementation of Bubble Sort be functionally equivalent to an implementation of Quck Sort?

Examples

1. If we compile a program with two different compilers for the same language, the result will (mostly) be two functionally equivalent programs.
2. Two compilers for the same language **need not** be functionally equivalent.

A detection strategy by Wheeler

Goal: We want to test a compiler A. Want to detect possible bugs "learned" by the compiler (in the sense of Thompson)

A detection strategy by Wheeler

Goal: We want to test a compiler A. Want to detect possible bugs "learned" by the compiler (in the sense of Thompson)

Requires: An *independent* compiler T (non-collusion betewen compiler A and compiler T).

Naming

Let S_A be the source code of compiler A and E_A its executable.

Let T be a compiler independent of A, with executable E_T.

Diverse double compiling

1. Compile S_A using E_A to get an executable X.
2. Compile S_A using E_T to get an executable Y.
3. Compile S_A using X to get an executable V.
4. Compile S_A using Y to get an executable W.
5. Compare V and W bitwise.

Observe: X and Y will be different binaries, but functionally equivalent.

Diverse double compiling (Step 1 & 2)

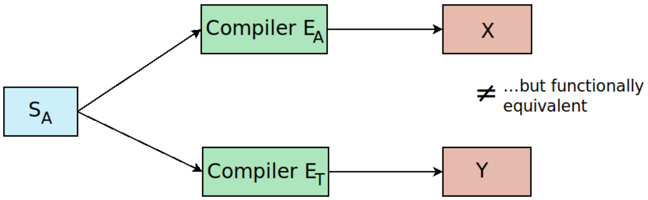

Figure 2: Step 1 & 2 of DDC

Diverse double compiling (Step 3, 4 and 5)

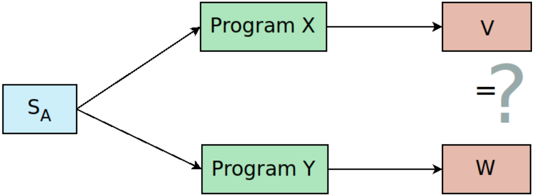

Figure 3: Step 3, 4 and 5 of DDC

Conclusions

- Should a ‘trusting trust’ type attack be part of our threat model?
- Thompson argues that at some point one must trust the people behind the software.
- Wheeler’s diverse double-compiling strategy gives guarantees under some assumptions (non-collusion).

Vulnerabilities

OWASP Top 10

- A1:2017-Injection
- A2:2017-Broken Authentication
- A3:2017-Sensitive Data Exposure
- A4:2017-XML External Entities (XXE)
- A5:2017-Broken Access Control
- A6:2017-Security Misconfiguration
- A7:2017-Cross-Site Scripting (XSS)
- A8:2017-Insecure Deserialization
- A9:2017-Using Components with Known Vulnerabilities
- A10:2017-Insufficient Logging&Monitoring

Vulnerabilities and exploits

Definition

> *A **vulnerability** is a weakness in the computational logic (e.g., code) found in software and some hardware components (e.g., firmware) that, when exploited, results in a negative impact to confidentiality, integrity, OR availability.*

(From mitre.org)

Disclosure

When a vulnerability is found, one has a choice:

- Should the vulnerability be **publicly disclosed**?

Disclosure

When disclosing vulnerabilities further questions arise:

- How much detail to include?
- Should an exploit be included? (if available)
- Should there be an embargo period?

Disclosure

There is a spectrum of different stances:

- **No disclosure**: No details should be made public.
- **Coordinated disclosure**: Details can be disclosed after fixes made and embargo lifted.
- **Full-disclosure**: full details should be publicly disclosed, and arguing against an embargo.

CVE

CVE

Common Vulnerabilities and Exposures (**CVE**) is a database of software vulnerabilities. Maintained by **The Mitre Corporation** in USA.

The list has entries consisting of:

- **A unique number** (CVE–YYYY–XXXX) identifying the vulnerability
- A desciption
- At least one public reference

CVE example

ID: **CVE-2018-7492**

Description: A NULL pointer dereference was found in the net/rds/rdma.c __rds_rdma_map() functionin the Linux kernel before 4.14.7 allowing local attackers to cause a system panic and a denial-of-service, related to RDS_GET_MR and RDS_GET_MR_FOR_DEST.

References:

- MISC:http://git.kernel.org/···commit/?id=f3069c6d33···
- URL:https://xorl.wordpress.com/···/linux-kernel-rdma-null-pointer-dereference/
- DEBIAN:DSA-4187
- ...

CVE number assignment

Assigning the CVE numbers is taken care of by the **CVE Numbering Authorities** (CNAs), which each have **different scopes**. These include:

- The Mitre Corporation (Primary CNA)
- Distributed Weakness Filing Project (For open-source projects)
- Many corporations (Google,Microsoft,Intel,Netflix,$\cdots$)

What is CVE used for?

CVE allows referencing vulnerabilities **accross systems**:

- Easier than referencing product/version/description:
  - **Easy**: CVE-2018-7492
  - Difficult: "That NULL pointer dereference in net/rds/rdma.c in Linux before 4.14.7."
- Easy to **track** vulnerability fixes:
  - From links we quickly find which Debian or Ubuntu packages contain the fixes.
- Provides a **quick way to look up vulnerabilities** for a given piece of software.

CVE numbers are often reported by vulnerability scanners which finger-print running services.

CVSS

Common Vulnerability Scoring System (**CVSS**) is a system for assigning a **score to a vulnerability**.

Includes three kinds of metrics:

- **Base metrics**, intrinsic properties
- **Temporal metrics**, changes over the vulnerability life-time
- **Environmental metrics**, specific to the environment of the software.

CVSS results in sevaral scores on a scale from **0–10**, based on a vector of metrics.

CVSS

Two different versions of CVSS are commonly used:

- Version 2
- Version 3

Link to the specification of version 3 on syllabus page.

Base metrics in CVSS Version 2

- Access vector:†
  - Local
  - Adjacent network
  - Network
- Attack complexity (High/Medium/Low)
- Authentication (Multiple/Single/None)

†: Version 3 adds "physical"

Impact metrics in CVSS Version 2

Rated on a scale of None/Partial/Complete impact:

- Confidentiality
- Integrity
- Availability

Temporal metrics in CVSS Version 2

The following metrics change over time:

- Exploitability
- Remediation level
- Report confidence

Exploitability

Exploitability is measured on a the scale:

- Unproven
- Proof-of-concept
- Functional
- High

Remediation level

Remediation level is measured on the scale

- Official fix
- Temporary fix
- Workaround
- Unavailable

Report confidence

Report confidence is measured on the scale

- Unconfirmed
- Uncorroborated
- Confirmed

CVSS example

Impact

CVSS v3.0 Severity and Metrics:

Base Score: 5.5 MEDIUM
Vector: AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:N/A:H (V3 legend)
Impact Score: 3.6
Exploitability Score: 1.8

Attack Vector (AV): Local
Attack Complexity (AC): Low
Privileges Required (PR): Low
User Interaction (UI): None
Scope (S): Unchanged
Confidentiality (C): None
Integrity (I): None
Availability (A): High

CVSS v2.0 Severity and Metrics:

Base Score: 4.9 MEDIUM
Vector: (AV:L/AC:L/Au:N/C:N/I:N/A:C) (V2 legend)
Impact Subscore: 6.9
Exploitability Subscore: 3.9

Access Vector (AV): Local
Access Complexity (AC): Low
Authentication (AU): None
Confidentiality (C): None
Integrity (I): None
Availability (A): Complete
Additional Information:
Allows disruption of service

Figure 4: CVE-2018-7492

Next time:

- CWE
- Tools