# Software Testing and Attack Injection

Ibéria Medeiros
Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

# Motivation

- Operating system facilities, such as the kernel and utility programs, are typically assumed to be reliable. In our recent experiments, we have been able to crash 25-33% of the utility programs on any version of UNIX that was tested.

  ⇨ Barton P. Miller, Lars Fredriksen e Bryan So, An Empirical Study of the Reliability of UNIX Utilities, CACM, Dec. **1990**

- The authors built a tool called *fuzz*
  - ☞ there was a thunderstorm, one of them was accessing a Unix machine from home using a dialup connection, causing packets to be corrupted and then some command line tools to crash
  - ☞ the tool reproduced conditions they found by accident

# Some Initial Thoughts

- There are an infinite number of possible ways that an application could fail, and organizations always have limited testing time and resources. **Be sure time and resources are spent wisely!**

- Try to **focus on the security holes that are a real risk to your business**! Try to contextualize risk in term of *the application* and *its use cases*

- **Tools do not make software secure!** They help **scale the process** and **help enforce policy**

# Software testing and security

- Software testing is a vast topic, but our focus is on finding security related problems

- Testing: evaluate software by observing its execution

  ☞ some people use different terms

  * **Static testing** – done without executing the software (we are particularly interested in *symbolic execution* and *static analysis*)

  * **Dynamic testing** – testing while running the software (we are interested in *fuzzing* and *attack injection*)

Other names you may encounter : *SAST – Static application security testing*; *DAST – Dynamic application security testing*

# Several kinds of testing

- Acceptance testing: verify that the sw satisfies its requirements
- System testing: verify if the sw satisfies its architectural design
  - ☞ assuming that the components satisfy their specification
- Module testing: verify if *sw modules* satisfy their specification
  - ☞ E.g., C modules, Java classes
- Unit testing: check if the *sw units* satisfy their specification
  - ☞ E.g., functions, methods
- Integration testing: verify the conformance of interfaces (i.e., if they are compatible), assuming modules are correct
- Regression testing: verify if changes made to sw did not impact its correctness

These tests are carefully performed to check that all *requirements* are correctly implemented by the system.

# Testing – challenges

- Observability: easiness of observing the sw behavior
  - ☞ getting the outputs, the effects in the environment and other sw/hw components
  - ☞ if low, it is difficult to obtain test results

- Controllability: easiness of providing inputs to the sw
  - ☞ if low, it is difficult to run the tests

- Coverage: capability of a test set to find all sw bugs
  - ☞ if low, only *parts* **or** *certain types* of bugs of the sw end up being tested

# Security vs Traditional testing

- Does traditional testing allow the discovery security bugs?
  - ☞ e.g., reflection vulnerabilities that allows a XSS attack?
- Functional testing: traditionally, testing aimed at checking that the sw does what it should do
  - ☞ focus mostly on verifying functional properties, i.e., that the sw does what it should
  - ☞ the previous list of kinds of testing shows exactly this objective: acceptance, system, integration, …
- Security testing: check that the sw does not do what it should not do
  - ☞ these requirements can be specified the same way as functional requirements, but often they are not specified
  - ☞ *Example for XSS*: input provided by a *client should not be reflected* to the client without adequate validation/sanitization

# Example with sendmail

- Functional testing: verify that messages are correctly delivered

- Security testing: verify if
  - ☞ debug command that provides increased functionality is disabled in the production environment
  - ☞ the email process is not running as root
  - ☞ the mailbox is not created with privileges that allow other user users to read the email

- Both: verify if message header allows buffer overflows
  - ☞ Functional: sw should reject input in incorrect formal
  - ☞ Security: obvious

# Security testing phases

1. Enumerate the ***attack surface***
2. Use ***attack modeling*** to prioritize the tests to be carried out
   - ☞ optional
3. Define ***tests that will be carried out***
   - ☞ based on the application requirements *and/or* common attack checklists
4. ***Execute the tests***
   - ☞ injecting inputs and monitor the sw behavior
5. Given the test result, perform ***code review*** to find the vulnerabilities in the program

# Definition of the tests

- Black box testing
  - ☞ Tests derived from **external** description of the software
  - ☞ Simplest form of deriving the tests
  - ☞ Often used by "security experts"
- White box testing
  - ☞ Tests derived from the **source code**
  - ☞ Harder but better coverage
  - ☞ Used by companies that develop the sw
- Gray box testing = white + black
  - ☞ Aims at the best of both worlds: simplicity of 1$^{st}$ expanded with coverage of 2$^{nd}$

Some people use "*white box testing*" to mean *static code analysis*, while "*black box testing*" is used for *fuzzing*

# FUZZERS

# Fuzzers

- Origin: story at the beginning of lecture
  - ☞ thunderstorm, then fuzz tool to inject random input

- Today fuzzing is often used for security testing
  - ☞ many recent vulnerabilities were found using fuzzing
  - ☞ basically they brute force the application with erroneous input to try to discover if it fails
    - – inputs are generated randomly, eventually through mutations of promising test cases
  - ☞ in many cases, monitoring is very simple (or inexistent)
    - – a test case is effective because the program crashed
  - ☞ attempts to run as many tests as possible within a short period of time
    - – each test case should take little time and take advantage of parallelism

# A minimal fuzzer - sharefuzz

- *sharefuzz*
  - ☞ simple fuzzer that tests environment variable usage (<100 locs)

- Method of insertion
  - ☞ Creates a shared library that

  1. is specified in LD_PRELOAD env. variable, thus always loaded
  2. redefines *getenv()*, overriding the original one, to always return a long list of identical characters for any environment variable (except DISPLAY) – by default returns 11500 A's

- Detection
  - ☞ the application is vulnerable if it crashes (no monitoring)

Note: may need access to program code to recompile with dynamic library

# Types of fuzzers

- In terms of *knowledge* of the target
  - ☞ Thin fuzzers – simple tools with little knowledge or assumptions about the app tested; send random invalid input to the target
  - ☞ Fat fuzzers – can generate input that is syntactically valid (accepted by the parser) but irregular to test how the target handles it; better coverage
- In terms of *specialization* of the application
  - ☞ Specialized – implemented for a specific type of application, or network protocol, or file format; can be made more “intelligent”
  - ☞ Generic – can be applied to a large spectrum of targets (*fuzzer frameworks*)
- In terms of *access to the code* of the application
  - ☞ Black – no access, and all tests simply go through the interface
  - ☞ Grey – ability to compile, eventually adding instrumentation code
  - ☞ White – the execution of the application is emulated, looking for conditions in the input that would cause a failure

# How to generate test cases (inputs)?

- Random fuzzing
  - ☞ generates random inputs (or at least parts of the input)
- Recursive fuzzing
  - ☞ iterating though all combinations of characters from an alphabet
  - ☞ Example: URL followed by 8 hexadecimal digits → try all combinations of the 8 digits
- Replacive fuzzing
  - ☞ iterating though a set of predefined values, called **<u>fuzz vectors</u>**
  - ☞ Example: look for XSS vulnerabilities by providing the inputs
    - >"><script>alert("XSS")</script>&
    - ";!--"<XSS>=&{()}
- Often there is a mixture of the above methods, using <u>mutations</u> on the fuzz vectors, adding random data, etc.

# Elements for fuzz vectors

- Think about known attacks: BOs, SQL inj., XSS,…
- Metacharacters
  - ☞ HTML  <  >
  - ☞ SQL  -  ;  ’  ”
  - ☞ OS  .  /  %00  *  |  ‘  ’
  - ☞ Web server  ../   %00
  - ☞ C / C++  %00
- Examples
  - ☞ Long string: 10K A’s
  - ☞ Very long string: 100K A’s
  - ☞ Format strings %n%n%n…
  - ☞ Paths ../../../../../../../../../../../../etc/passwd
  - ☞ Paths ../../../../../../../../../../../../etc/passwd%00
  - ☞ Odd characters: metacharacters and others
  - ☞ XSS  <script>alert(document.location);</script>

OWASP Testing Guide 4.0 provides a list of fuzz vectors for attacks against web apps: XSS, SQL inj., BOs, IOs, format string, LDAP inj., XPATH inj., XML inj.

# Example fuzz vectors

- To test for SQL injection

  ’ or 1=1--

  " or 1=1--

  ’ or 1=1 /*

  or 1=1--

  ’ or ’a’=’a

  " or "a"="a

  ’) or (’a’=’a

  Admin’ OR ’

  ’ or 1 in (select @@version)--

  ’ union all select @@version--

# SPIKE - A fuzzer framework

- Fuzzer framework with a few pre-defined fuzzers and allows the addition of others
- Data representation is important in this type of fuzzers
  - ☞ user has to provide input data
  - ☞ the fuzzer does variations on this data
  - ☞ data may be binary or strings

- SPIKE uses a data structure called a *spike* for data representation, and then provides a number of functions to connect and send data to the target

# SPIKE (cont)

- *Spike* = sequence of structures to represent sequence of bytes
  ☞ a structure can represent a <u>block size</u> *or* a <u>queue of bytes</u>
- The programmer has to define a *spike* that will be modified and injected

- Example of a spike

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01 02 03 04");
s_string_variable("ola!");  // fuzz this field
s_block_end("somepacketdata");
```

☞ stores in the queue
  - a word (4 bytes), big endian, with the length of the block
  - the 4 bytes *0x01020304*
  - add mutations on string "ola!"

# AFL – American Fuzzy Loop

- Aims to fuzz diverse programs in a fast and robust way, by **instrumentation** and **genetic algorithms** to automatically discover interesting test cases (inputs)
  - ☞ which trigger new internal states in the targeted binary
  - ☞ substantially improve the coverage of the fuzzed code
  - ☞ eventually cause the crash of the program
- Can start with a range of inputs, such as an empty input or specific test cases that should be tried
- Outputs interesting test cases that caused misbehaviors (plus some statistics)
- Supports the execution of several fuzzer instances in parallel that collaborate to cover larger portions of the code

# AFL - Instrumentation

- Adds a small amount of code to each branch (edge) of the program at compilation time

value generated randomly

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

64kB shared memory region

every byte set corresponds to a hit for (branch_src, branch_dst)

preserve directionality of tuples (A ^ B != B ^ A) and identity of tight loops (A ^ A != B ^B)

- Keeps efficiently but in an imprecise way, information about which parts of the code are run when processing some input
  - ☞ allows the detection that a new test case causes different blocks of the program to be executed
  - ☞ supports the discovery that a new test case causes a certain block to be run a different number of times

# AFL – Pool of Test Cases

- AFL keeps a pool of test cases
  - ☞ new test cases are created by mutating existing test cases
  - ☞ each previously tried test case has an associated a global map of tuples (branch_src, branch_dst)
  - ☞ test cases that either contain new tuples (i.e., the exploration of novel code blocks) or tuples with (relevant) differences in hit counts are added to the pool for further fuzzing
  - ☞ Otherwise, they are discarded
- Mutation
  - ☞ test cases in the pool selected for mutation and future processing with a certain probability that depends on the input size and execution latency
  - ☞ example mutations are: sequential bit flips of varying lengths and step-overs; addition/subtraction of small integers; etc …

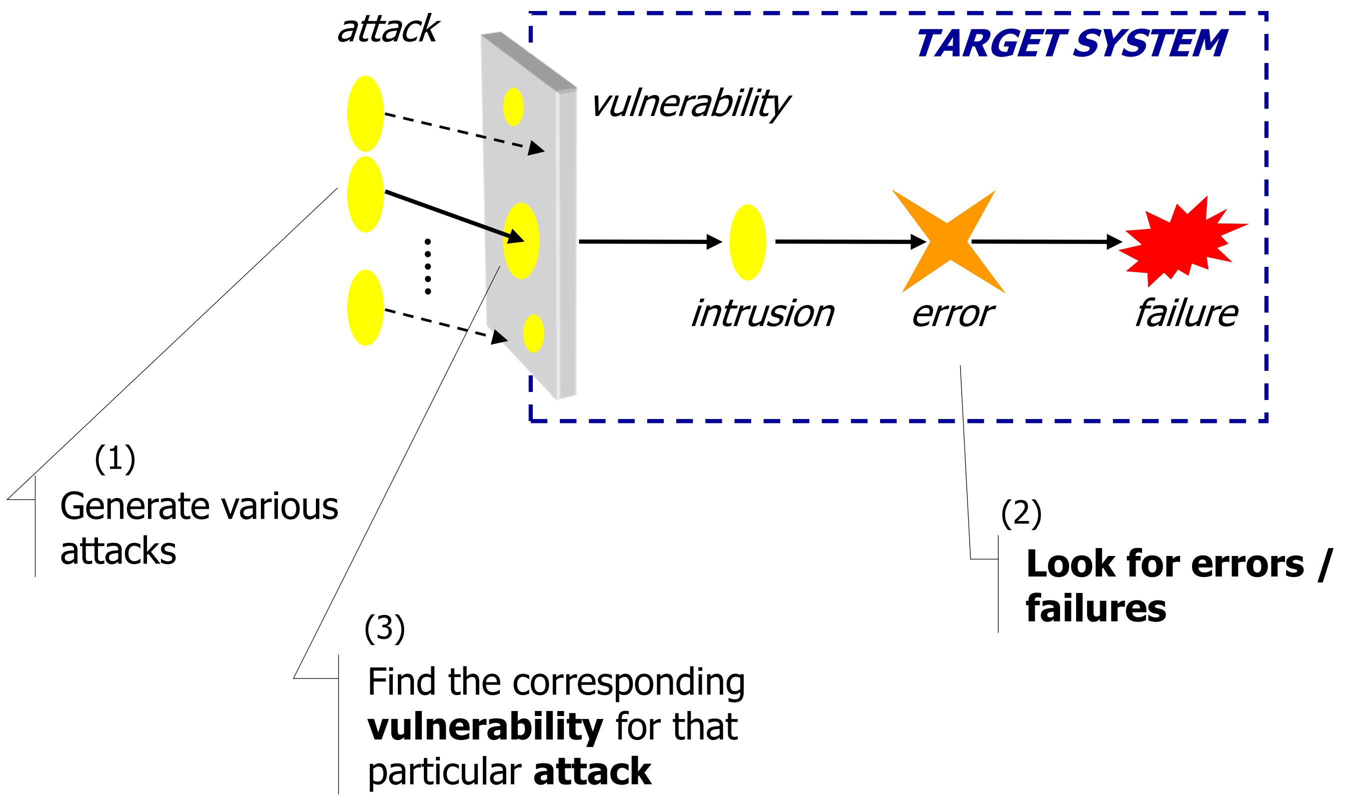
# Other Fuzzers

- SPIKE comes with several demo fuzzers
  - ☞ Webfuzz – combination of tools for web fuzzing
  - ☞ MSRPC, SunRPC
- Mangle, FileFuzz – fuzzes file formats
- Holodeck – fuzzes several parameters in Windows
- WSFuzzer (OWASP) – fuzzer for web services
- Project PROTOS has fuzzers for several protocols: WAP, LDAP, SNMP, SIP,...
- Bunny the Fuzzer (Google) – C programs
- Sulley – a fuzzer framework, like SPIKE

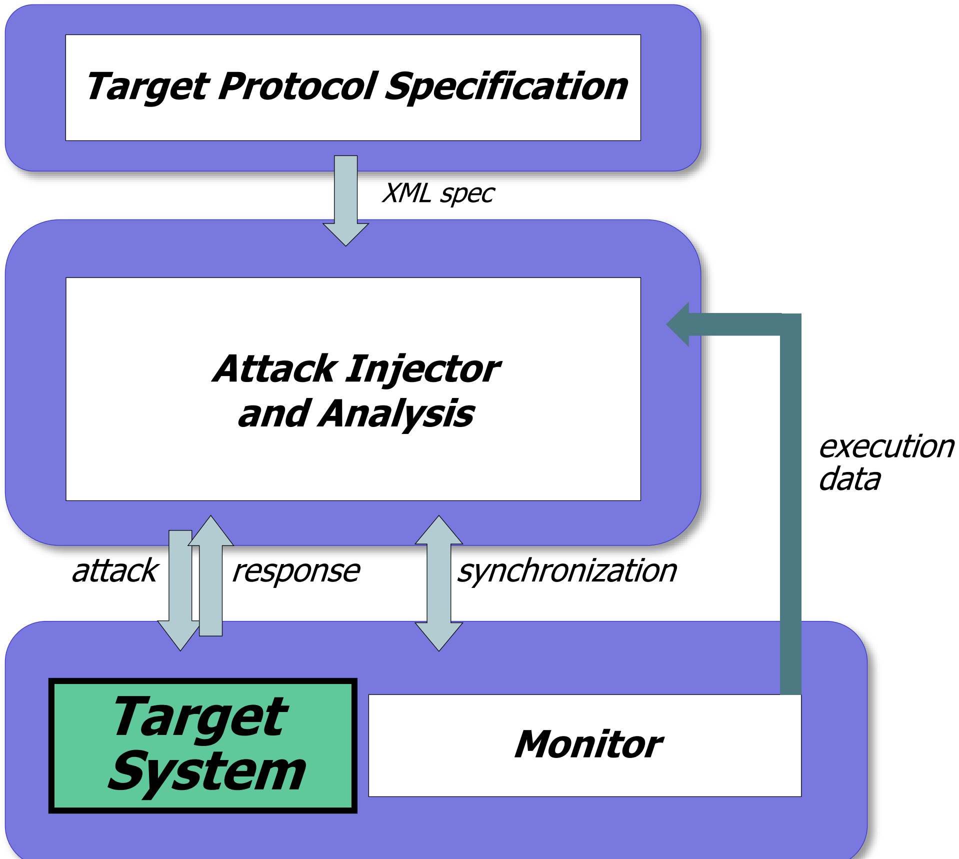
# ATTACK INJECTION

# Attack Injection

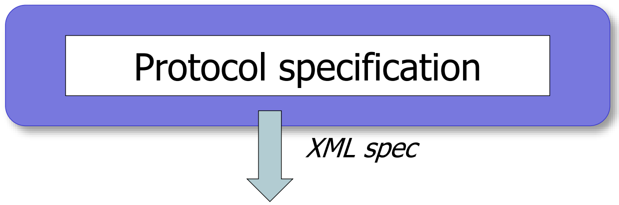
- The attack-intrusion-vulnerability model



# A generic architecture



# Target Protocol Specification

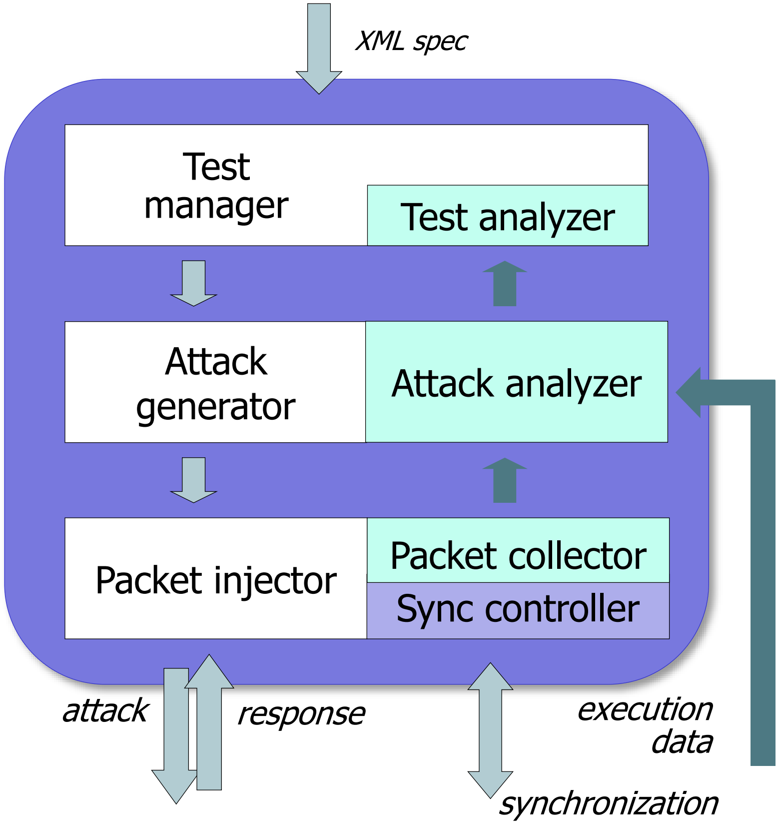



- Collect knowledge about the target so that more effective attacks can be generated
  - ☞ States, messages, data types
  - ☞ Transition between protocol states
  - ☞ Creation of valid/invalid protocol packets

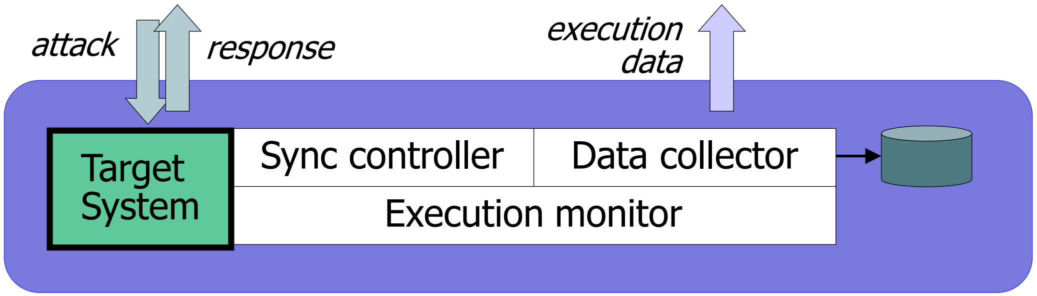
- How do we get the specification?
  - ☞ GUI for generating specification (XML file)
  - ☞ automatically with protocol reverse engineering

# Attack Injector

- Controls the injection process
  - test manager: defines test classes based on the protocol specification
  - test analyzer: analyzes the results of the attacks
- Creates the attacks
  - attack generator: creates the attacks
  - attack analyzer: uses results and input to know if the attack was successful
- Inject (malicious) packets
  - packet injector: sends malicious packets
  - packet collector: stores the packets injected and received as results
  - synchronization controller: cooperates with the monitor to prepare the target application for an attack



# Target System and Monitor




- Target System = *target application* + execution environment

- Monitor depends on the operating system, but has as main functions the *setup of the experiment* and *the collection of data about the execution*
  - ☞ Process: system calls that are invoked; interception of Unix signals
  - ☞ Resource usage supervision
    - – Memory usage: number of pages (total), number of virtual memory pages, number of pages in RAM
    - – CPU usage: CPU time used in user and kernel mode
    - – Disk: number and size of read/write accesses

# Inject what?

- Injection is composed of <u>two parts</u>
  - ☞ take the target to a particular state of its execution
  - ☞ inject some input that tests the behavior
- <u>Ideally</u>: all possible combinations of inputs
- Consider a simple Web application with
  - – 10 forms, where a form has 10 fields
  - – every field with space for 50 characters, each can take 62 different characters (A-Z, a-z, 0-9)
  - ☞ Number of valid inputs: 10 x 10 x 62^50 ~= 4E91
    - – 1 second for each gives 1E84 years!

☞ *and **invalid inputs** must also be tested!*

# Example tests: Syntax

- Take a specification of the format of a message composed by several fields
- Attacks that infringe the syntax of the protocol
  - ☞ Original packet spec = $field_1 + field_2 + field_3$
  - ☞ Packets with field permutations, added fields, and removed fields :
    - – $attack_1$ packet = $field_1 + field_3$
    - – $attack_2$ packet = $field_2 + field_3$
    - – $attack_3$ packet = $field_1 + field_2$
    - – $attack_4$ packet = $field_1 + field_1 + field_2 + field_3$
    - – …..

Ended up being very simple and not very effective!

# Example tests: Value

- Attacks with packets with erroneous data
  - ☞ Original packet spec = $\text{field}_1$ + $\text{field}_2$

    *$\text{field}_1$ - 4 bytes integer (0..100)*

    *$\text{field}_2$ - word (string ending in space)*
  - ☞ Packet fields with *dangerous* values

    $\text{field}_1$: -1, 0, 100, 101, -100

    $\text{field}_2$: very long word, non-printable ASCII chars
  - ☞ Example: {-1, ABSFS%EGER$GE#WTRE}

    {-1, OHJJAEROG$JOT$%Y$YHWEGHJHW$#F}
  - ☞ Packet fields with *malicious tokens*

    $\text{field}_1$: -1, 0, 100, 101, -100

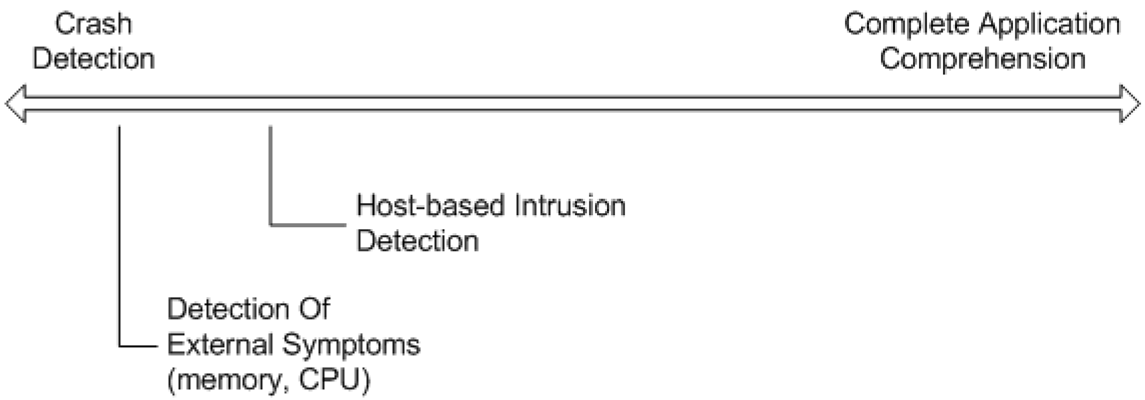
    $\text{field}_2$: malicious tokens + payload combinations
  - ☞ Example: {-1, ASDOJWSGV%sERJGERR}

    {-1, FVREIOREVGOERVJSD“””OGEORBVEAR}

# Monitoring

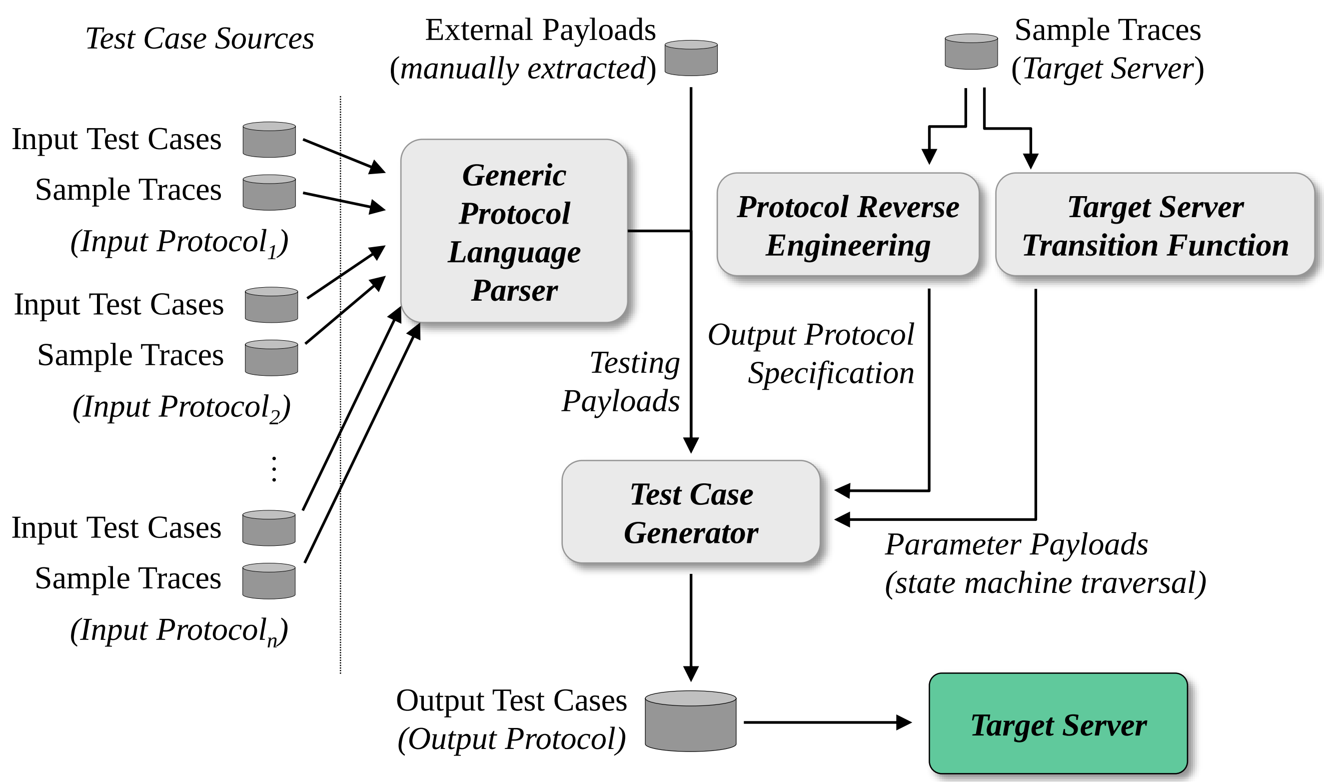
- Objective: to find out if an attack activated a vulnerability
  - ☞ Difficult! To be done precisely it would require a complete knowledge and understanding of the internal state of the target
  - ☞ Think about a buffer overflow: how do we know if it exploited a vulnerability? What about a resource exhaustion vulnerability?

- Specter of monitors



# Smarter Test Generation

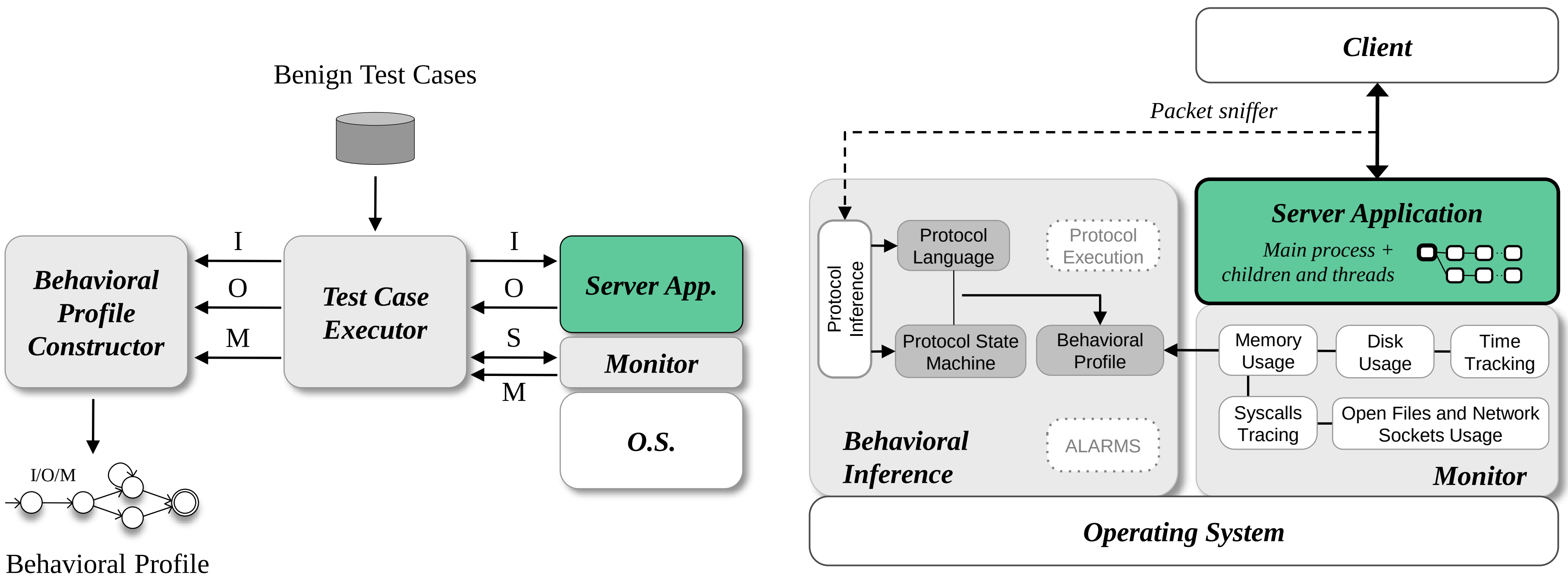
- Take advantage of existing test cases to [generate new tests]() that
  - cover novel functionality *and*
  - different protocols



# Smarter Monitoring and Analysis (1)

- Use protocol reverse engineering techniques to build a **model of the correct execution** of the target

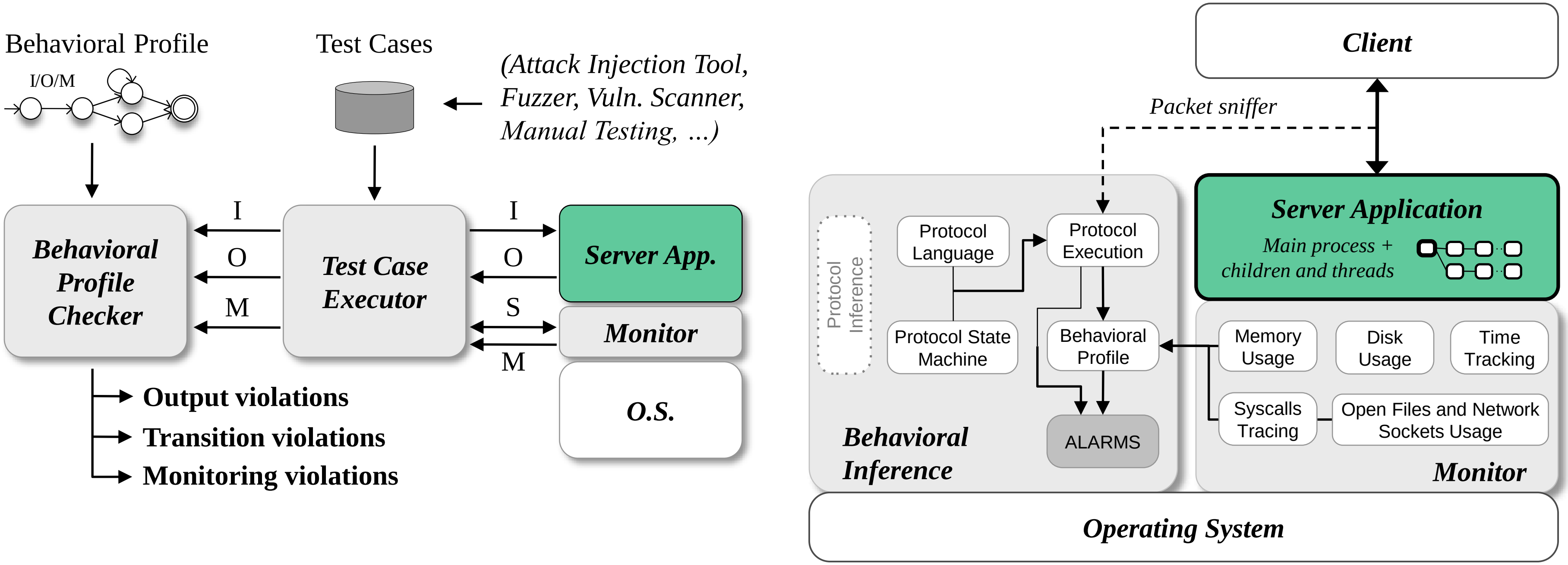
## LEARNING PHASE



# Smarter Monitoring and Analysis (2)

- Discover vulnerabilities by looking for discrepancies between the behavioral model and the observed execution

## TESTING PHASE



# VULNERABILITY SCANNERS

# Vulnerability scanners

- Classical search for vulnerabilities in computer systems
  - ☞ experiment different attacks that *exploit certain already known vulnerabilities* (described in a database)
  - ☞ often, they provide hints on how to fix the discovered flaws
  - ☞ Examples: OpenVAS, Nessus, LANguard, COPS, SATAN

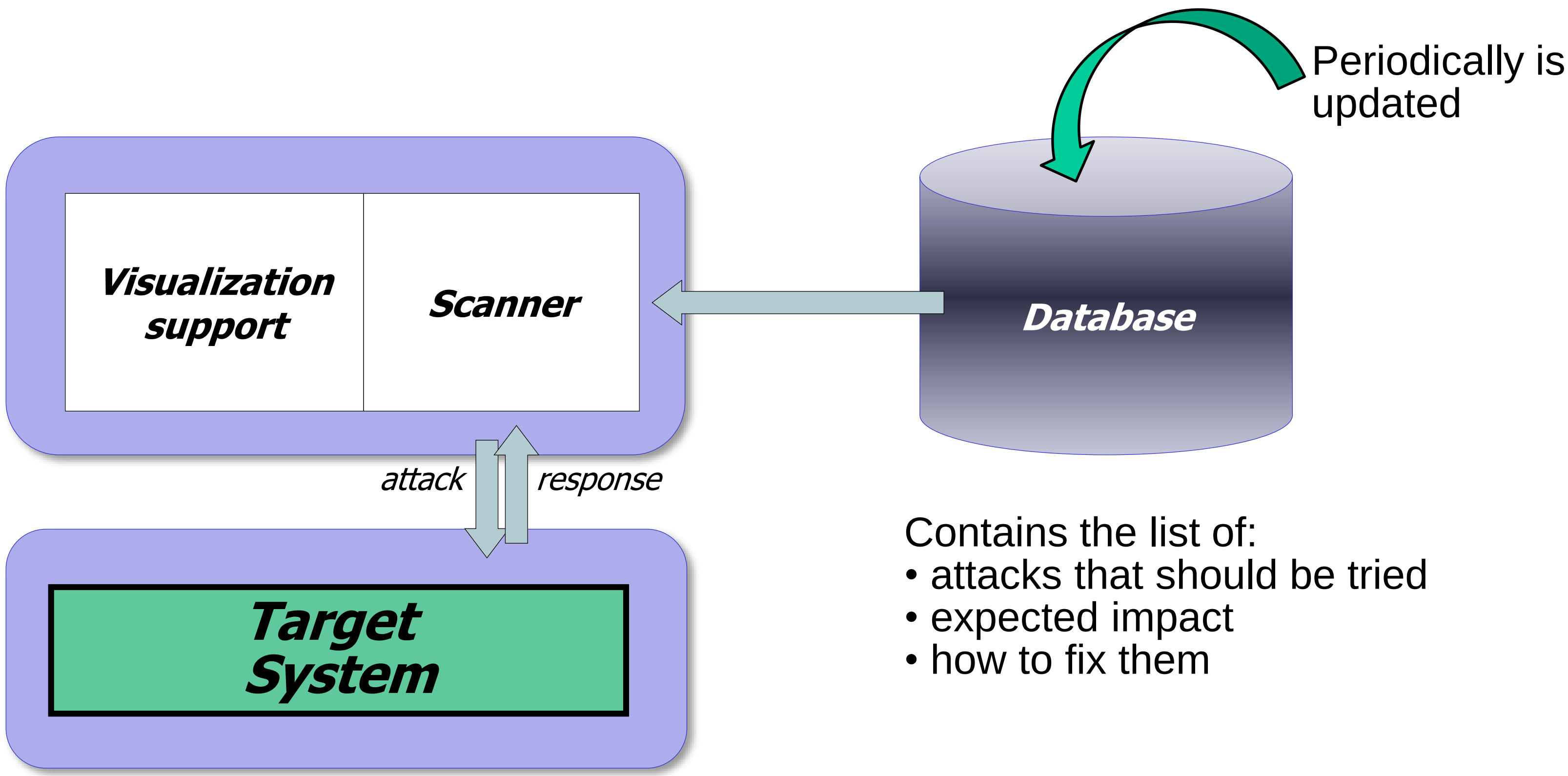
- Web vulnerability scanners – for web applications, more recent
  - ☞ also called *web application scanners, web application security scanners, web application vulnerability scanners*
  - ☞ *we will only consider these ones …*

# Web vuln.scanners – NIST

- NIST specified a set of basic requirements
  - Identify all of the types of vulnerabilities in their list
  - Report an attack that demonstrates the vulnerability
  - Have an acceptably low false positive rate
  - Create report in a format compatible with other tools - XML (optional)
  - Indicate remediation tasks (optional)
  - Use normalized names for vulnerability classes, e.g., CWE (optional)

- Mandatory support for the detection of 14 classes of vulnerabilities
  - XSS, SQL injection, command injection, XML injection, HTTP response splitting, file inclusion, direct reference to objects, CSRF, improper information disclosure, broken authentication / weak session management, session fixation, insecure communication, failure to restrict URL access
  - *But* ***not all*** *vulnerabilities* (impossible), only all classes

# A generic architecture



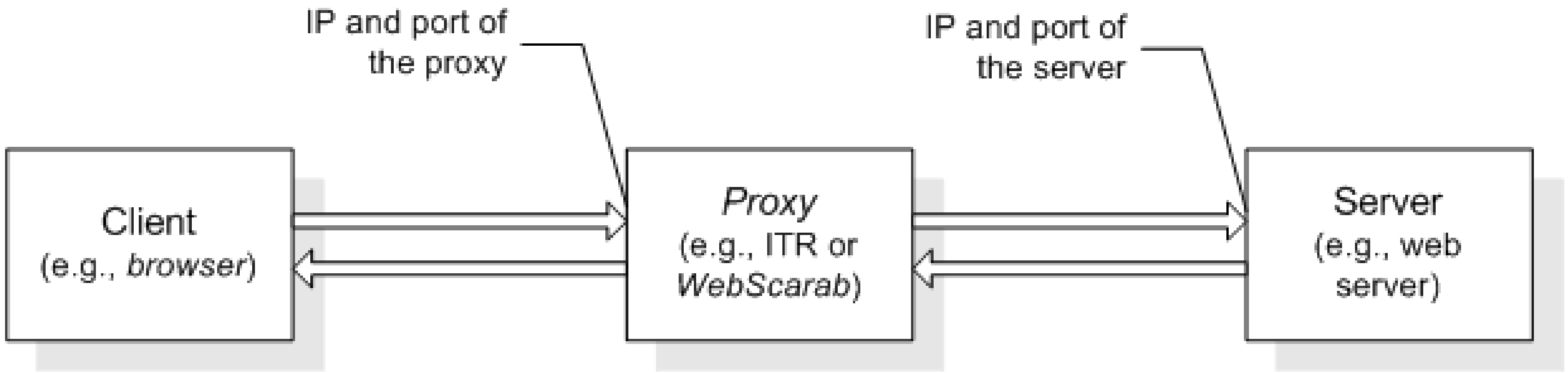
# Web vuln.scanners vs AI/fuzzers

- Separation is not entirely clear
- Web vulnerability scanners also “inject attacks”, like attack injectors and fuzzers
- Web vulnerability scanners run database of specific attacks, while the others are (more) random
- Web vulnerability scanners are commercial tools, while the others tend to be free/open
  - ☞ Example: Acunetix WVS, IBM Rational AppScan, HP WebInspect

# PROXIES

# Proxies

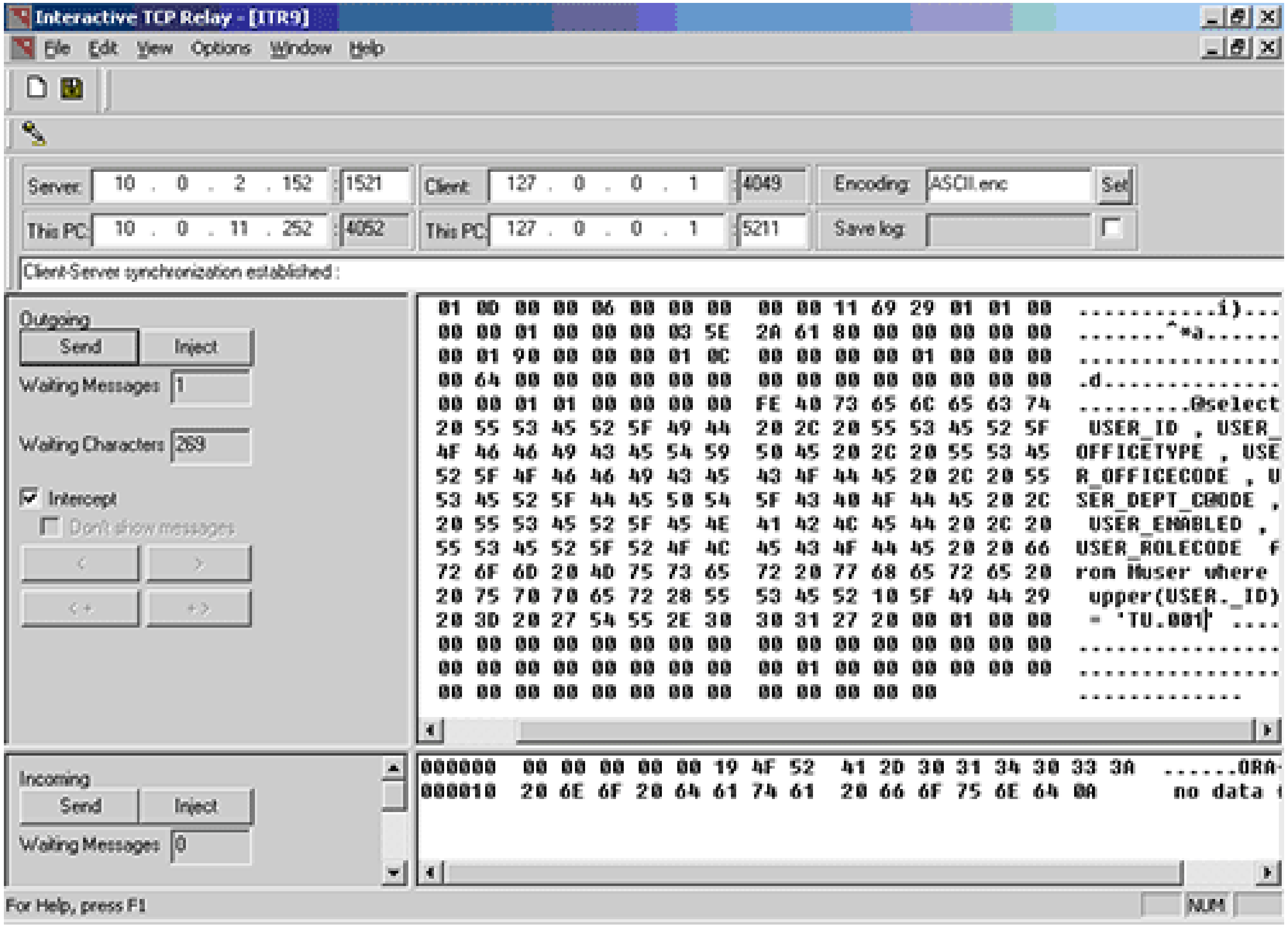
- The discovery/definition of the application protocol can be difficult to accomplish
- Special *proxies* can be used in this context
  - ☞ no need to obtain a protocol specification …
  - ☞ simply modify the messages in the network
- Architecture



# A minimal proxy - ITR

- ITR: Interactive TCP Relay – allows to test client/server applications that communicate with TCP
  - ☞ started with the port to listen to and the server IP and port
  - ☞ the client is given the ITR's IP instead of the server's
  - ☞ can log the communication
- Interception
  - ☞ ITR can intercept outgoing / incoming traffic
  - ☞ data is buffered until the user indicates that it should be sent
  - ☞ while buffered, the user can manually modify data using a hex editor
  - ☞ additional data can be injected directly in the connection

# ITR




data in hex
and ascii

# ZAP

- Contains a more sophisticated capability of performing fuzzing automatically in a web request

- Go to the lab and try it out …

# Bibliography


- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017      (see chapter 13)


Other references: