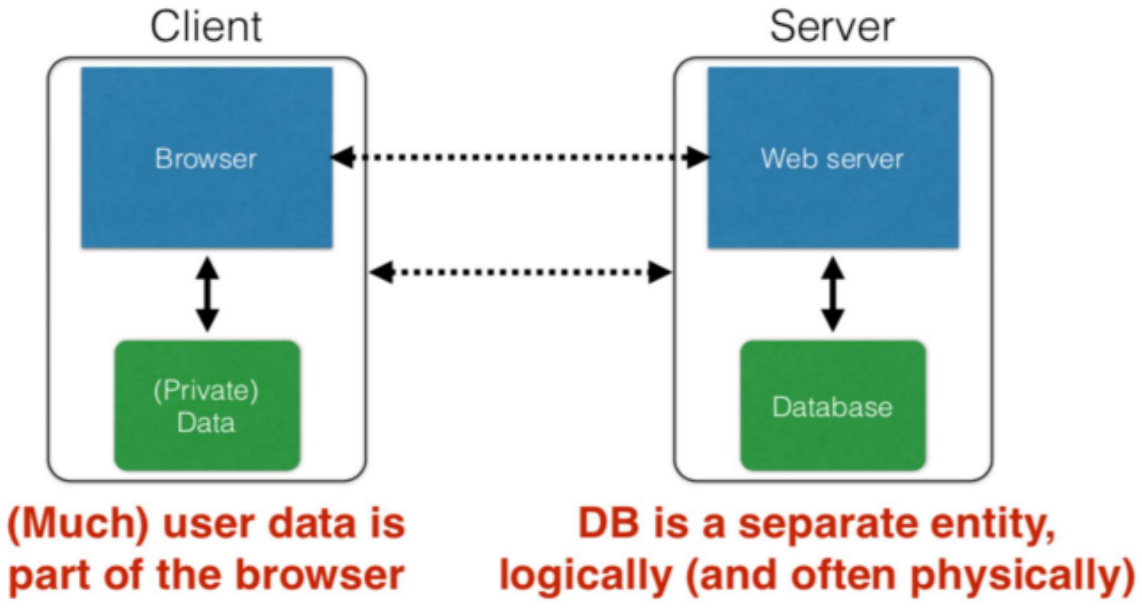
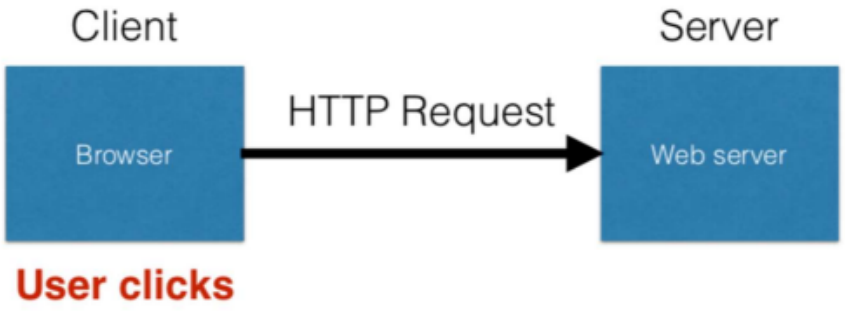
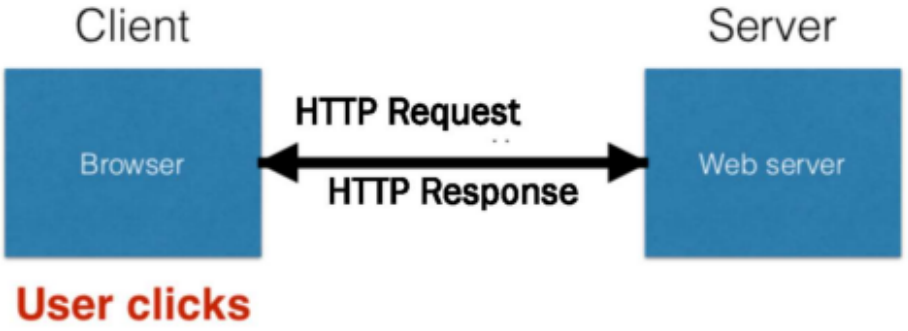
IS U4 NOTES



HTTP - Application-layer



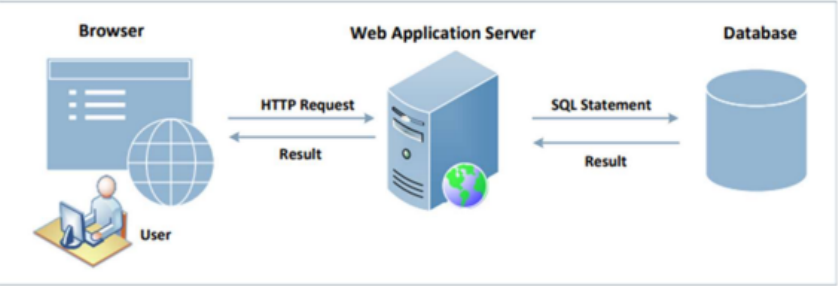
- **Requests contain**:
  - The **URL** of the resource the client wishes to obtain
  - **Headers** describing what the browser can do
- **Request types** can be **GET** or **POST**
  - **GET**: all data is in the URL itself (no server side effects)
  - **POST**: includes the data as separate fields (can have side effects)



- **Responses** contain:
  - **Status** code
  - **Headers** describing what the server provides
  - **Data**
  - **Cookies** (much more on these later)
    - Represent s*tate* the server would like the browser to store on its behalf

Interacting with Database in Web Application

☞ A typical web application consists of three major components:



☞ SQL Injection attacks can cause damage to the database.

- As we notice in the figure, the users do not directly interact with the database but through a web server.
- If this channel is not implemented properly, malicious users can attack the database.

Getting Data from User

- The request shown is an HTTP GET request, as the method field in the HTML code specified the get type.
- In GET requests, parameters are attached after the question mark in the URL. Each parameter has a name=value pair separated by "&".
- In HTTPS, the format is similar, but the data is encrypted.
- Once this request reaches the target PHP script, the parameters inside the HTTP request will be saved to an array `$_GET` or `$_POST`.

```php
<?php
   $eid = $_GET['EID'];
   $pwd = $_GET['Password'];
   echo "EID: $eid --- Password: $pwd\n";
?>
```

How Web Applications Interact with Databases

1. Connecting to MySQL Database

- PHP programs connect to the database server before conducting a query on the database.
- The following code example shows how to create a database connection using `new mysqli(...)` along with its four arguments:

```
function getDB() {
   $dbhost="localhost";
   $dbuser="root";
   $dbpass="seedubuntu";
   $dbname="dbtest";

   // Create a DB connection
   $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);
   if ($conn->connect_error) {
         die("Connection failed: " . $conn->connect_error . "\n");
   }
   return $conn;
}
```

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// Create connection
$conn = new mysqli($servername, $username, $password, $dbna
me);

// Check connection
if ($conn->connect_error) {
  die("Connection failed: " . $conn->connect_error);
}
echo "Connected successfully";
?>
```

2. Constructing and Executing Queries

- PHP constructs the query string and then sends it to the database for execution.
- The channel between the user and the database creates a new attack surface for the database.

This structured explanation covers how data is transferred from the user to the server, how PHP interacts with a MySQL database, and the potential security risks associated with this interaction.

Launching SQL Injection Attacks

- Everything provided by the user becomes part of the SQL statement.

- It is possible for a user to change the meaning of the SQL statement.
- The intention of the web app developer by the following is for the user to provide some data for the blank areas.
- Assume that a user inputs a random string in the password entry and types “EID5002’#” in the eid entry. The SQL statement will become the following:

```
SELECT name, salary, SSN FROM employees WHERE eid='EID500
2'#' AND password='randomstring';
```

- Everything from the `#` sign to the end of the line is considered a comment.
- The SQL statement will be equivalent to the following:

```
SELECT name, salary, SSN FROM employees WHERE eid='EID500
2';
```

- The above statement will return the name, salary, and SSN of the employee whose EID is EID5002 even though the user doesn’t know the employee’s password. This is a security breach.

Let’s see if a user can get all the records from the database assuming that we don’t know all the EID’s in the database.

- We need to create a predicate for WHERE clause so that it is true for all records.

Problem

- Assume that a database only stores the sha256 value for the password and eid columns.
- The following SQL statement is sent to the database, where the values of the $passwd and $eid variables are provided by users.
- Does this program have an SQL injection problem?

Answer:

- It still has an SQL injection problem.
- For example, we can let eid be “x, 256)’ OR 1=1 #”.

Problem

- The following SQL statement is sent to the database, where $eid and $passwd contain data provided by the user. An attacker wants to try to get the database to run an arbitrary SQL statement. What should the attacker put inside $eid or $passwd to achieve that goal?

```
$Sql = "SELECT * FROM employee WHERE eid=’$eid’ and passwor
d=’$passwd’";
```

Answer:

- We can put the following in $eid:

```
'; DROP TABLE employee; --
```

This would make the SQL query:

```
SELECT * FROM employee WHERE eid=''; DROP TABLE employee; -
-' and password='$passwd'
```

And this would drop the entire employee table from the database.

Static Analysis

Current Practices for Software Assurance

Testing:

- Testing is conducted to ensure that a program runs correctly for a given set of inputs.

Benefits:

- Concrete failure during testing proves there’s an issue in the code.
- Once an issue is identified, it aids in fixing the problem.

Drawbacks:

- Testing can be expensive and difficult.
- Achieving code path coverage is hard; it's difficult to cover all code paths.

- Even after extensive testing, there's no guarantee that the program is bug-free.

Static Analysis Goals:

- Bug finding: Identify code that the programmer wishes to modify or improve.
- Correctness: Verify the absence of certain classes of errors.

Code Auditing:

- Convince someone else your source code is correct.
- Benefit: Humans can generalize beyond single runs.

Drawbacks of Code Auditing:

- Expensive, hard, and no guarantees.

These practices help in ensuring the quality and reliability of software, but each has its limitations and challenges.

Example

Consider the following C function that prints a message to a specified file descriptor without performing any error checking:

```
void printMsg(FILE* file, char* msg) {
    fprintf(file, msg);
}
```

If either argument to this function is null, the program will crash. Programming defensively, we might check to make sure that both input parameters are non-null before printing the message, as follows:

```
void printMsg(FILE* file, char* msg) {
    if (file == NULL) {
        logError("attempt to print message to null file");
    } else if (msg == NULL) {
        logError("attempt to print null message");
    } else {
        fprintf(file, msg);
```

```
    }
}
```

A security-conscious programmer will deprive an attacker of the opportunity this vulnerability represents by supplying a fixed format string.

```
void printMsg(FILE* file, char* msg) {
    if (file == NULL) {
        logError("attempt to print message to null file");
    } else if (msg == NULL) {
        logError("attempt to print null message");
    } else {
        fprintf(file, "%.128s", msg);
    }
}
```

Static Analysis for Secure Development

Introduction

- Static analysis: What, and why?

Basic Analysis

- Example: Flow analysis, It tracks the flow of information through the program, checking for inconsistencies and errors.

Increasing Precision

- Context-, flow-, and path sensitivity.

Scaling it up

- Pointers, arrays, and intricate information flows.

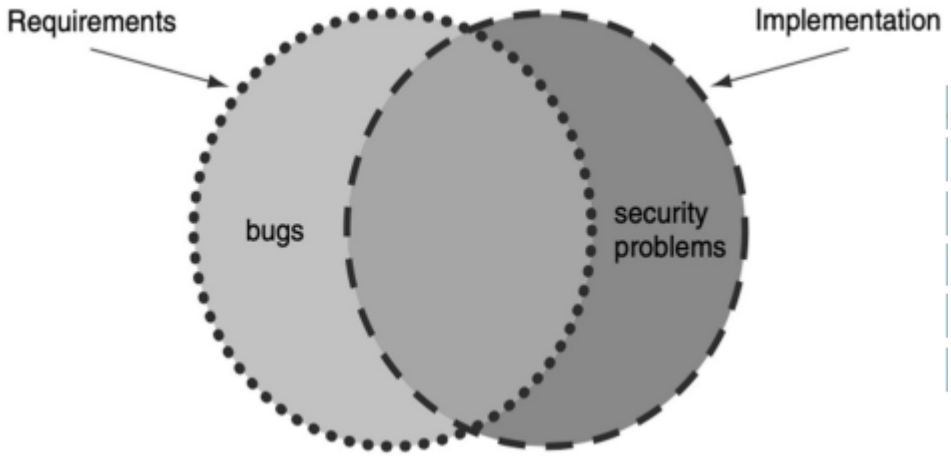
Security Features != Secure Features

Consider this misguided quote from BEA's documentation for WebLogic [BEA, 2004]:

- Since most security for Web applications can be implemented by a system administrator,
    - Application developers need not pay attention to the details of securing the application unless there are special considerations that must be addressed in the code.

- For programming custom security into an application, WebLogic Server application developers can take advantage of BEA-supplied Application Programming Interfaces (APIs) for obtaining information about subjects and principals (identifying information for users) that are used by WebLogic Server.
    - The APIs are found in the weblogic security package.

The Quality Fallacy



Security problems are frequently "unintended functionality" that causes the program to be insecure. Whittaker and Thomson describe it with the diagram [Whittaker and Thompson, 2003].

- **Reliable software does what it is supposed to do.**
- **Secure software does what it is supposed to do, and nothing else.**

Static Analysis

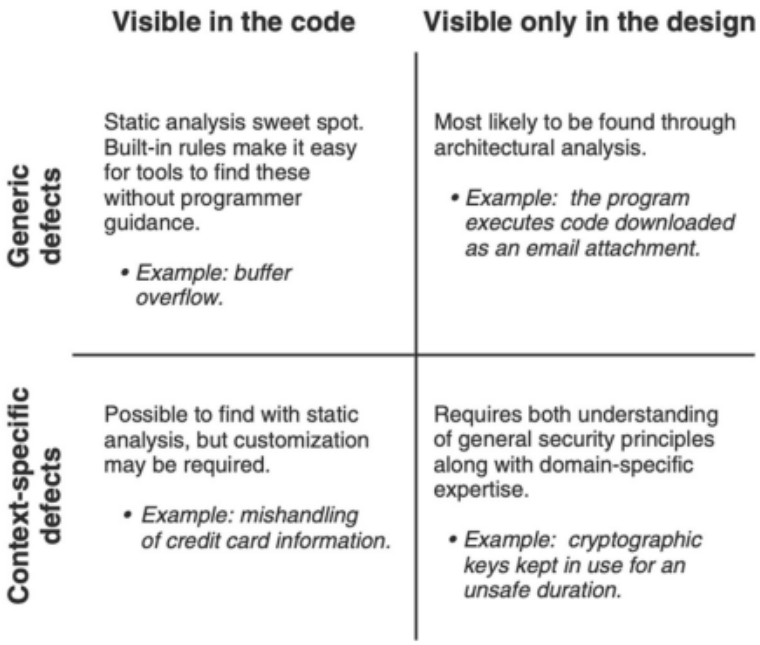
- Analyze the program's code without running it!
    - In a sense, we are asking a computer to do what a human might do during a code review.
- Benefit is (much) higher coverage.
    - It's like having a super-powered code reviewer who never sleeps, never gets tired, and can review thousands of lines of code in seconds!
    - Sometimes all of them, providing a guarantee.
- Drawbacks:

- Can only analyze limited properties.
- May miss some errors or have false alarms & be time-consuming to run.

Impact

- Thoroughly check limited but useful properties!
  - Eliminate categories of errors!
  - Developers can concentrate on deeper reasoning!
- Encourages better development practices.
  - Develop programming models that avoid mistakes in the first place.
  - Encourage programmers to think about and make manifest their assumptions – Using annotations that improve tool precision!
- Seeing increased commercial adoption.

Classifying Vulnerabilities



Security – an integral part of Software

Treating the symptom:

- Focusing on security after the software is built is suboptimal.

Treating the cause:

- Focusing on security early, with activities centered on the way the software is built.
  - Security Requirements
  - Misuse cases
  - Threat Modelling
  - Static Code Analysis

Classifying Vulnerabilities

The Seven Pernicious Kingdoms

1. Input Validation and Representation
2. API Abuse
3. Security Features
4. Time and State
5. Errors
6. Code Quality
7. Encapsulation

Array Bounds, Interrupts Testing

- Errors typically occur on uncommon paths or with uncommon input.
- Difficult to exercise these paths.

Inspection

- Non-local and thus easy to miss.
- Array allocation vs. index expression.
- Disable interrupts vs. return statement.

The Paradox of Perfect Static Analysis

- Useful static analysis is perfectly possible, despite:
  - **Non-termination: The analyzer might enter a loop and never terminate.**

    - **False alarms: Claimed errors are not really errors.**
    - **Missed errors: Absence of error reports doesn't mean the code is error-free.**
- Non-terminating analyses are confusing, so tools tend to exhibit only false alarms and/or missed errors.

The Art of Static Analysis

- **Design Considerations in Analysis - Analysis design tradeoffs:**
    - Precision
    - Scalability
    - Understandability
- Observation: Code style is important!
    - Aim to be precise for "good" programs.
- False alarms viewed positively: reduces complexity.

Solving Problems with Static Analysis

- Type checking
- Style checking
- Program understanding
- Program verification
- Property checking
- Bug finding
- Security review

Other Challenges

- Taint through operations!
    - Tainted a; untainted b; c= a+b - is c tainted? (yes, probably)
- Function pointers!
    - What function can this call go to?
    - Can flow analysis compute possible targets?
- Struct fields!

    - Track the taintedness of the whole struct, or each field?
    - Taintedness for each struct instance, or shared among all of them (or something in between)?
- Arrays!
    - Keep track of the taintedness of each array element, or one element representing the whole array?

Other Kinds of Analysis

- Pointer Analysis (“points-to” analysis)
    - Determine whether pointers point to the same locations.
    - Shares many elements of flow analysis.
- Data Flow Analysis!
    - Invented in the early 1970s. Flow-sensitive, tracks.
    - It is flow-sensitive, meaning it takes into account the order in which operations occur.
- Abstract interpretation!
    - Invented in the late 1970s as a theoretical foundation for data flow analysis and static analysis generally.
    - Associated with certain analysis algorithm.

Tainted Flow Analysis

- The root cause of many attacks is trusting unvalidated input.
    - Input from the user is tainted.
    - Various data is used, assuming it is untainted.
- Examples expecting untainted data:
    - Source string of strcpy (≤ target buffer size).
    - Format string of printf (contains no format specifiers).
    - Form field used in constructed SQL query (contains no SQL commands).

Analysis Problem

- No tainted data flows:
  - For all possible inputs, prove that tainted data will never be used where untainted data is expected.
  - `untainted` annotation: indicates a trusted sink.
  - `tainted` annotation: an untrusted source.
  - No annotation means: not sure (analysis figures it out).
- A solution requires inferring flows in the program:
  - What sources can reach what sinks.
  - If any flows are illegal, i.e., whether a tainted source may flow to an untainted sink.

Flow Sensitivity

- Our analysis is flow insensitive:
  - Each variable has one qualifier which abstracts the taintedness of all values it ever contains.
- A flow-sensitive analysis would account for:
  - Variables whose contents change.
  - Allow each assigned use of a variable to have a different qualifier.
  - E.g., $\alpha 1$ is x's qualifier at line 1, but $\alpha 2$ is the qualifier at line 2, where $\alpha 1$ and $\alpha 2$ can differ.
- Could implement this by transforming the program to assign to a variable at most once.
  - Called static single assignment (SSA) form.

Why not flow/path sensitivity?

- Flow sensitivity adds precision, and path sensitivity adds even more, which is good.
- But both of these make solving more difficult!
- Flow sensitivity also increases the number of nodes in the constraint graph.
- Path sensitivity requires more general solving procedures to handle path conditions.

- In short: precision (often) trades off scalability!
    - Ultimately, limits the size of programs we can analyze.

Pen Testing

Penetration testing assesses security by actively trying to find exploitable vulnerabilities!

- Black hat activity (for a good purpose). Practitioners variously called red teams, tiger teams, etc.
- Can be applied at different levels of granularity:
    - Program (single process)
    - Complete application (communicating processes)
    - Network of many applications
    - Generally not libraries or incomplete pieces of code

Who, and how

- Pen testers employ ingenuity and automated tools.
- To rapidly explore a system's attack surface, looking for weaknesses to exploit.
- Typically carried out by a separate group within, or outside, an organization, separate from developers.
- Avoids tunnel vision: Same reason doctors tend to not treat themselves or their own families.
- Given varied access to system internals:
    - From no access, like outside attacker, to full access, like a knowledgeable insider.

History

- 1967 Ware Report
    - Task force of experts headed by Willis Ware of RAND Corp. formally assessed the security problem for time-sharing computer systems. Used term "penetration".
    - Read more

- 1970s: DOD penetration testing teams emerge to assess “real” security of government computer systems.
- Today: Penetration testing is expanding.
    - Popular with students, e.g., “CTF” competitions like picoCTF, GoogleCTF, etc.
    - Many companies can be contracted to do penetration testing for Certified Penetration Tester (CPT), certificates like EC-Council CEH, IACRB CPT, etc.

Types of Penetration Testing

1. Overt Penetration Testing (White box penetration testing)

- You work with the organization to identify potential security threats.

Advantages:

- Full access without blocks.
- Detection doesn’t matter.
- Access to insider knowledge.

Disadvantages:

- Don’t get the opportunity to test incident response.

2. Covert Penetration Testing (Black box penetration testing)

- Performed to test the internal security team’s ability to detect and respond to an attack.

Advantages:

- Test incident response.
- Most closely simulates a true attack.

Disadvantages:

- Costly, time-consuming, require more skill.
- Due to the cost of covert testing, most will target only one vulnerability, the one with the easiest access. Gaining access undetected is key.

Benefits

- Penetrations are certain and reproducible demonstrated by tests.

    - Not hypothetical!
- Applied to a whole component, not code fragments.
- No false alarms.
- "Feel good" factor.
- Produces evidence of real vulnerabilities that would otherwise have gone unfixed.
- Thus, results in a clear improvement to security.

Drawbacks

- Absence of penetrations is not evidence of security!
    - After fixing any issues, there may be others still lurking.
- Changes to the system necessitate a retest!
- Security is not compositional: a change to one component may render another component insecure.
- But changes are common!
    - Can be expensive to retest too frequently.
    - So must retest the entire system.

Pen Tester’s Bag of Tricks

- A pen tester approaches a target knowing:
    - The workings of the target domain (e.g., the web).
- How systems are built in that domain:
    - Protocols (e.g., HTTP, TCP, ...).
    - Languages (e.g., PHP, Java, Ruby, ...).
    - Frameworks (e.g., Rails, Dream Weaver, Drupal).
- Common weaknesses in the software/system:
    - Bugs (e.g., SQL injections, XSS, CSRF, ...).
    - Misconfigurations, bad design (e.g., default passwords, “hidden” files, ...).

Web hacking: A professional’s view

- 70% messing with parameters.
  - If the URL is http://tgt.com/buy?item=1&price=5.00, then change it to:
    - /buy?item=1&price=0.01
    - /buy?item=10&price=5.00
- Eric Eames of FusionX suggests:
  - Are client parameters (unwisely) trusted?
  - Is there XSS vulnerability?
- 10% default passwords:
  - Always research the default password and try it.
  - Works way more often than you'd think.
- 10% hidden files and directories:
  - Look through the manuals for clues.
  - Directory brute forcing.
- 10% other:
  - Authentication problems (bypass, replay, ...).
  - Insecure web services.
  - Configuration page gives away your root password.

Tools

- Pen testers use tools to:
  - Probe a target.
  - Gather information and test hypotheses about it.
  - Exploit a vulnerability (or attempt to).
- The tool depends on the goal and the target:
  - If an enterprise network, want to find, probe, and exploit machines, routers, topology, etc.
  - If a single machine, want to consider installed software, running programs, interesting files.

    - If a single program, want to explore and exploit possible inputs and interactions.

Nmap for Network Probing

- Nmap stands for “network mapper”.
    - Determines:
        - What hosts are available on the network,
        - What services (application name and version) those hosts are offering,
        - What operating systems (and OS versions) they are running,
        - What type of packet filters/firewalls are in use,
        - And more.
- Works by sending raw IP packets into the network and observing the effects.
- Free, open source and (commercial tools too).

Finding Hosts, Services

- Standard “ping” protocol.
- Nmap will ping a specified range of IP addresses.
- ICMP Echo Request and/or Timestamp request.
    - TCP SYN to port 443, TCP SYN/ACK to port 80.
        - Looking for running HTTPS or HTTP servers.
    - Other things, as determined by the operator.
- Protocol-specific UDP packets to particular ports.
- Probes to other TCP ports.
- Probes that elicit different responses on different OSes (“fingerprinting”).

Be stealthy

- A flurry of scanning activity may be detected.
- Control the rate of scanning to “work under the radar”.

Web Proxies

- Web applications are common pen testing targets.
- Web proxies sit between the browser and server.
- They:
  - Display exchanged packets.
  - Modify them as directed by the tester.
  - Some proxies have additional features for vulnerability scanning/exploitation, site probing, etc.

ZAP - OWASP – Zed Attack Proxy

- GUI-based inspection/modification of captured packets.
- Can set “breakpoints” to allow packets through until a certain condition is met.
- **Features:**
  - Active scanning: attempts XSS, SQL injection, etc.
  - Fuzzing: context-specific payloads.
  - Spider: explores a site to construct a model of its structure.
-

Metasploit

- Advanced open-source platform for developing, testing, and using exploit code.
- Boasts an extensible model through which payloads, encoders, no-op generators, and exploits can be integrated.
- **Scripting attacks:**
  - Probe remote site looking for vulnerable services.
  - Construct payload based on versions, other features.
  - Encode payload to avoid detection.
  - Inject payload.

    - Wait for shellcode to connect back; command prompt.

Metasploit UI

- `msfconsole`: interactive console for executing Metasploit commands.
    - Also web-based frontend and command-line interface.
- Supports probing and communications commands, payload construction (and encoding).
- Supports active (go get 'em) and passive (wait till they come to us) attacks.
- **Meterpreter**:
    - Command processor injected into the target, e.g., in the memory of a compromised process.
    - Permits the pen tester to probe more stealthily.
- `msfpayload`, `msfencode`: generate (stealthy) shellcode.

Burp Suite

- Burp is an essential offensive security tool used by a majority of professionals (including pentesters).
- Dedicated mainly to pentesting web applications.
- Modular tool that allows both manual and automated tests to be carried out, helping pen testers.

Kali

- Kali is a Linux distribution with many open-source pen testing tools installed and configured.
- Includes tools like:
    - Nmap, Zap, Metasploit, Burp Suite, and dozens more.
    - John the Ripper for password cracking.
    - Valgrind for dynamic binary analysis.
    - Reaver for Wifi password cracking.
    - Peepdf for scanning PDF files for attack vectors, and more.

Ethical Hacking

- Penetration testing tools are meant to reveal security vulnerabilities so they can be fixed, not so they can be exploited in the wild.
- However, people use tools for nefarious purposes. Don’t be one of them.

Fuzzing

What is Fuzzing?

- A kind of random testing.
- Goal: make sure certain bad things don’t happen, no matter what.
  - Crashes, thrown exceptions, non-termination.
- All of these things can be the foundation of security vulnerabilities.
- Complements functional testing.
- Test features (and lack of misfeatures) directly. Normal tests can be starting points for fuzz tests.

Kinds of Fuzzing

- **Black box**
  - The tool knows nothing about the program or its input.
  - Easy to use and get started, but will explore only shallow states unless it gets lucky.
- **Grammar based**
  - The tool generates input informed by a grammar.
  - More work to use, to produce the grammar, but can go deeper space.
- **Whitebox**
  - The tool generates new inputs at least partially informed by the code of the program being fuzzed.
  - Often easy to use, but computationally expensive.

Fuzzing Inputs

- **Mutation**
  - Take a legal input and mutate it, using that as input.

    - Legal input might be human-produced, or automated, e.g., from a grammar or SMT solver query.
    - Mutation might also be forced to adhere to grammar.
- **Generational**
    - Generate input from scratch, e.g., from a grammar.
- **Combinations**
    - Generate initial input, mutateN, generate new inputs.
    - Generate mutations according to grammar.

File-based Fuzzing

- Mutate or generate inputs.
- Run the target program with them.
- See what happens.

Examples: Radamsa and Blab

- **Radamsa** is a mutation-based, black box fuzzer.
    - It mutates inputs that are given, passing them along.

```
% echo "1 + (2• + (3 + 4))" | radamsa --seed 12 -n 4
5!++ (3 + -5))
1 + (3 + 41907596644)
1 + (-4 + (3 + 4))
1 + (2 + (3 + 4
%echo … | radamsa --seed 12 -n 4 | bc -l
```

- **Blab** generates inputs according to a grammar (grammar-based), specified as regexps and CFGs.

```
% blab -e '(([wrstp][aeiouy]{1,2}){1,4} 32){5} 10’
soty wypisi tisyro to patu
```

Example: Burp Intruder

- Burp automates customized attacks against web applications.

- Similar to SPIKE in allowing the user to craft the template of a request, but leave “holes” (called payloads) for fuzzing.
- Nice GUI front end.
- Integrates with the rest of the Burp Suite, which includes a proxy, scanner, spider, and more.