Chapter 33
FPVA Step 4, Component Analysis

Revision 1.0, September 2025.

Objectives

- Learn to think like an analyst.
- Understand the fourth step of FPVA, the component evaluation.
- Start finding vulnerabilities in real software.

33.1 Motivation

In the previous modules you learned about FPVA steps 1–3. The goal was to get the big picture of the system that you are assessing. That means understanding the architecture of the system, the different resources the system uses, the privilege levels for the different components, and who owns and accesses the system's resources. Once you have completed these first steps, you are ready to start looking for vulnerabilities. The motivation for these first steps was to focus your attention on the critical parts of the system so that you will be looking for vulnerabilities affecting the high value assets. This approach can help you to avoid wasting your time assessing parts of the system that are less likely to contain important vulnerabilities. Ideally you would have the time and resources to assess the whole system, but we must be realistic: that does not happen even in well funded efforts.

This chapter will guide you on how to start looking for vulnerabilities, keeping in mind that a vulnerability is identified as such only if you can build an exploit for it. So, we are talking about confirmed vulnerabilities instead of potential vulnerabilities. Once you find a vulnerability (congratulations), you will write a vulnerability report for the vulnerability you found, as explained in the next chapter on FPVA Step 5.

To look for the vulnerabilities we describe in this chapter, you will have to experiment with the system (messing around with it and hopefully breaking it) and inspect the source code. We suggest a variety of approaches for each of the different categories of vulnerabilities we cover, and we illustrate some of the results with vulnerabilities that we have found when applying FPVA to real systems. Note that the list of issues we suggest is just a starting point. Every system is different. However, after finishing looking for the problems discussed in this chapter, you will be ready to go beyond and look for vulnerabilities that are specific to the system you are assessing. So, let's get started.

33.2 Roadmap of Vulnerabilities Discussed in this Chapters

The rest of this chapter covers a variety of vulnerabilities that we have found in some of our real world software assessments. Each section is organized by the type of vulnerability found. In each section is a brief description of the vulnerability then a subsection for each high level approach (technique) used to find the vulnerability. Also included are example vulnerabilities that illustrate the vulnerability type and approach to finding it. The table below serves as a roadmap to the material covered in this chapter.

		High Level Approach								
		Permission inspection	Focused code inspection	Focused file inspection	Experiment with system	Run static analysis tools	Run dependency tools	Run fuzz tools	Message interception	Stress testing
Vulnerabilities	Authorization Issues	X	X		X					
	Issues Found by Tools					X	X	X		
	Cross Site Request Forgery (CSRF)		X		X				X	
	Abusing Authentication Mechanism		X	X						X
	Denial of Service									X
	Too Much Information		X		X					
	Exposed traffic								X	
	Injections: SQL, XSS, Code, Command, XML, Path name		X		X					X
	Buffer Overflow		X		X					X

33.4 Authorization Issues

Authorization refers to what a user is allowed to do in a system. A system is vulnerable if resources (such as files) have incorrect permissions. Privilege escalation allows a user to perform operations beyond their intended privilege level, and that is another example of an authorization problem.

High level approach: Permissions inspection

Check permissions for log files and for configuration files. The permissions must match the authorized user/group. No user should be able to read files they are not authorized to read, which is a violation of confidentiality. In the same way, no user should be able to write files that they are not authorized to read, which is a violation of integrity. Given that log files may contain sensitive information such as passwords, or session identifiers, reading such files could have serious consequences.

> *Example of a vulnerability from Open XDMoD:* The XDMoD software logged the session cookies for every request it received in the process of checking their validity. The log file was globally readable and could allow an attacker to read the cookies and then hijack active sessions.

Check permissions for executable files (application binaries). You need to pay special attention to which users are allowed to execute such files. Also check that the UID/GID for the running processes are the intended ones. If a process is running as user root, it will be able to access any resources in the system.

> *Example of a bug from Open onDemand:* When requesting Open onDemand to launch a new program, the file that contained the script to launch the program did not have execute permission, so the launch failed.

> *Example of a bug from Open XDMoD:* Open XDMoD has a collection of utility scripts that are used only by the system administrator. However, the files containing these scripts had incorrect permissions that would allow any user to execute them.

High level approaches: Focused code inspection, Experiment with the system

Try to find where privilege escalation can happen. For example, check if you can tamper with the parameters of a `setuid`[1] call. Also check if you can tamper with files that contain passwords or other kinds of credentials. Also check if a user can attack other users running on the same system. This type of attack includes a user accessing (for example removing) processes or

[1] `https://man7.org/linux/man-pages/man2/setuid.2.html`

container images that belong to another user. The processes from the different users should be isolated. Even with virtual machines or `cgroups`[2], it is important to make sure that there is sufficient isolation. File systems should be properly mapped and isolated.

> *Example of a vulnerability from Singularity:* Gaining root access inside of a container allowed for root access on the underlying host machine. A user executing as root inside the container can mount the host root file system, allowing it to modify the /etc/passwd file and set the root password on the host, allowing privilege escalation on the host machine.

> *Example of a vulnerability from Custos:* Any user with access to the Custos REST API can change the password of any other user. This is a consequence of not having a mechanism in place to verify the source of a password reset request.

> *Example of a vulnerability from Custos:* In Custos, a tenant refers to an application that is controlled by Custos. Such tenant applications have an administrator and regular users. Any valid user could update the metadata of any tenant in Custos. This metadata includes the tenant administrator information; thus, any valid user can make themselves the administrator of any tenant in Custos, or simply deny service to the valid administrator.

> *Example of a vulnerability from HTCondor:* Different programs from different users running on the same host belonged to the same user ID. Therefore, any of those programs could kill the programs belonging to other users.

Abuses to authorization can result from improper validations. Check that before executing any operation, a server performs the necessary validations to ensure that the operation is executed on behalf of a user who is authorized. Validations on the client side are important, but it is even more important that those validations also happen on the server side. Software systems should have common code to do the checking to prevent multiple implementations of the code performing the validation, which is more error prone and difficult to maintain.

> *Example of a vulnerability from CATOS_WebIP:* Improper validation allows users to view information belonging to other users. The client interface restricts a user to viewing only items belonging to that specific user, however the server does not perform that validation. Instead, the server simply searches the database for matches without any sanitization.

> *Example of a vulnerability from CATOS_WebIP:* Improper input validation in the server allows attackers to illegally download, upload,

[2] `https://en.wikipedia.org/wiki/Cgroups`

overwrite, or delete files throughout the server's file system. Operating System file permissions limit this vulnerability to affecting only files that are accessible to the owner of the server process (e.g., if the server process is started by the SYSTEM Windows user, then all files are vulnerable).

33.5 Issues Found by Tools

Both static analysis tools and dependency tools can provide useful information in finding vulnerabilities in a program. In addition, dynamic techniques such as fuzz random testing can expose execution errors. When using such tools, you will have to determine if the reported problems can lead to vulnerabilities.

Static analysis tools scan the source code or bytecode of a program and report on weaknesses found in the code. While some of these reported weaknesses may be vulnerabilities, the tools can also generate many false positives, so you will need to carefully analyze the output of such tools.

Dependency analysis tools will tell the analyst about security issues affecting the software supply chain.

Finally fuzz testing tools help debugging the system, and some of the bugs they find may be security related.

Note that running just one static analysis tool or one dependency analysis tool is not enough, and can give a false sense of security. In Module 6, we elaborate on static analysis and dependency tools and in Module 7, we address fuzz testing.

High level approach: Run standard static analysis, dependency, and fuzz testing tools

Modern software is not built from scratch but on top of usually complex software stacks. You need to find outdated and vulnerable dependencies in the software supply chain, using dependency check tools. Also, it is important to use automated assessment tools to find vulnerabilities in the code. Note that this is not a silver bullet that will find all the vulnerabilities in your code, but it is a good starting point. Furthermore, it is recommended to use Fuzz testing to make your system crash or hang, and then use a debugging tool to identify the problem.

Example of a vulnerability from Custos: The code that implements Custos' Core Services and Integration Services have multiple dependencies with known vulnerabilities. Several of these vulnerabilities are considered critical.

33.6 Cross Site Request Forgery (CSRF)

High level approach: Message interception/Communication monitoring, Experiment with the system

Communications should be encrypted, as we well know. The first step is to check if HTTPS is used so messages are encrypted. Instead, if HTTP is used, attackers can use tools to intercept and modify traffic, and therefore be able to submit any requests they want.

Try to submit a request to the server without using the client attack surface (visible fields). This means attacking the application using a REST client to access parts of the application that are not accessible through the UI. Watch the browser traffic and try to replicate a request using `curl`.

> *Example of a vulnerability from Custos*: Any user with access to the Custos REST API can change the password of any other user. This is a consequence of not having a mechanism in place to verify the source of a password reset request. The Custos web server, as with any web server, receives requests, and those requests can come from the client user interface or from a command line tool such as `curl`. The server needs to **validate** the requests it receives before serving them.
>
> *Example of a vulnerability from Custos*: In Custos, a tenant refers to an application that is controlled by Custos. Such tenant applications have an administrator and regular users. Any valid user could update the metadata of any tenant in Custos. This metadata includes the tenant administrator information; thus, any valid user can make themselves the administrator of any tenant in Custos, or simply deny service to the valid administrator. This vulnerability comes from the fact that the service responsible for updating the metadata does not check that the session ID corresponds to the administrator of the tenant.

High level approach: Focused code inspection.

You need to understand how sessions are managed in your system. Weak session management results in attackers being able to generate fake requests. So, you need to check how session identifiers are generated. Nonces is a mechanism that prevents CSRF, therefore you need to check if nonces are used in the requests/responses. In addition, check if sessions time out

33.7 Abusing the Authentication Mechanism

High level approach: Multiple attempts (stress attempts)

Perform a brute force attack to try to get another user's credentials (password), and check if there is a limit to the number of attempts for relevant operations, such as login attempts.

Example of a vulnerability from Custos: An unauthorized user can find valid user credentials through a dictionary or brute force attack on the login endpoint of the Custos REST API. There is no limit to the number of invalid login attempts that can be made by a user, thus any unauthorized user can make unlimited login attempts until they find a set of valid credentials. Additionally, the ability to execute unlimited login attempts creates the potential for a denial of service attack. Each unsuccessful login attempt generates logs on the Custos server that, if not handled appropriately, can fill the disk partition.

High level approaches: Focused code inspection, Stress input, Focused file inspection

Check if a user can impersonate another user, for example by getting access to some other user's token or certificate that grants access. Furthermore, check if any information used to generate credentials is unprotected, for example stored in environment variables.

Example of a vulnerability from CREAM: A malicious user can, under the right conditions, replace another user's proxy certificate with their certificate. This proxy certificate is used for the user's access to a program execution service running on another computer. New requests submitted by the regular user to this execution service will use the malicious certificate and the regular user's programs will execute under the identity of the attacker giving the attacker full control over the programs and the data used by those programs. This vulnerability was caused by weak permissions on the directory storing proxy certificates.

Example of a vulnerability from Tapis: Any local Tapis user can decode their respective user JSON Web Tokens (JWTs) and modify them in such a way that they can impersonate other users and services.

You need to understand what protocol is used for authentication. If it is not a well-known protocol, try to dissect it to find flaws. Even if it is a well known protocol, you need to check that the implementation of the protocol is also a well-known one. Otherwise, there are chances to find vulnerabilities in the implementation.

If tokens are used, check if the system implements token rotation. This allows us to detect token theft. Also investigate if passwords and authorization codes are being hashed.

33.8 Denial of Service (DoS)

High level approach: Stress load

An attacker causing a DoS will prevent the system from being available to valid users. To examine if the system is vulnerable to a DoS, try to exhaust the available resources, such as filling up the free space on the file system partition by continuously writing to a log file, or spawn processes continuously. As part of this, conduct stress tests involving repeated requests and more demanding requests, and assess the use of resources (such as memory, disk, or CPU).

Check for "leftover" processes, such as zombie processes[3] or a container running in the background. If those are found, try to generate many of them.

> *Example of a vulnerability from Singularity:* Singularity allows users to run containers in the background using the singularity `run` command and the shell "&" operator. A user can execute a container that, when brought back into the foreground, can only be killed by the user or root from another window.

> *Example of a vulnerability from Open XDMoD:* Every time a request was made to Apache, an entry was logged to a specific log file. By repeatedly sending requests to Apache (even invalid ones), an attacker can fill up the free space on the file system partition causing a DoS.

33.9 Too Much Information (TMI)

33.9.1 High level approach: Focused code inspection, Experiment with the system

You need to inspect the code for exception handling and check if:

A. Exceptions are correctly used. The system must include exception handling for dealing with abnormal conditions. Make sure that the error messages printed do not disclose information about internals of the system. It is also important to check that messages intended for the debugging stage of the software are removed from the production version of the product.

B. For SQL queries, check if too many tuples are returned for certain queries. For example, when checking if there is a password match in a table, only one entry should be returned.

> *Example of a bug from Open XDMoD*: When authenticating a request, either from the session cookies or a provided user name and password, XDMoD performs a query to select all entries from the database that match the given credentials. This operation returns a list of tuples (table rows), not necessarily one. In both checking the session cookies and

[3] A zombie process is one that has exited but its kernel state has not yet been cleaned up because no parent process has checked its exit status. You can find more details at `https://en.wikipedia.org/wiki/Zombie_process`.

verifying the user's password, the only verification performed on the number of types is whether the list is empty or has greater than zero tuples. If it has at least one tuple, then the first one is assumed to be the intended one. If there was an error in the authentication database or if there was some other related attack (such as a SQL injection), the query might incorrectly return more than one tuple.

In addition, you will need to experiment with the system and see the error messages you get after the system executes a request that fails. Also pay attention to the case where an error message was intended for debugging purposes, but ended up in the released (and deployed) software.

33.10 Exposed Network Traffic

High level approach: Message interception/communication monitoring

Start by checking if the protocol used is HTTP instead of HTTPS. If it is HTTP, intercept the network traffic between different components. For that you will need to use a tool to read the unencrypted traffic (attack to confidentiality), to modify/inject traffic (attack to integrity), and to destroy traffic (attack to availability).

Example of a vulnerability from Open XDMoD: The default configuration of Open XDMoD does not encrypt HTTP traffic. This misconfiguration allows attackers to monitor all traffic between the server and the client. As a result, passwords submitted on login are sent in plain text, and can be stolen.

Example of a vulnerability from Open onDemand: It was possible to intercept the unencrypted traffic between some of the processes that implement the functionality of Open onDemand. Because the Open onDemand configuration at the time of the assessment had these connections on internal networks, that was not an issue of immediate concern. However, if a future configuration change moves one of the involved processes out of the same protection environment, then the associated connection would become vulnerable.

33.11 Injections: SQL, XSS, Code, Command, XML, Path Name

High level approach: Stress input, Focused code inspection, Experiment with the system

- *SQL injections*: Try to abuse input fields (attack surface), and include SQL queries in an input field. Also inspect the path from the attack surface to potential impact surfaces. To do that follow the data flow in the code, starting at the attack surface.

- *XSS*: Try to abuse input fields (attack surface), and include JavaScript code in the input fields.
- *Code injections*: Identify if any user supplied input ends up being executed. To find such cases, inspect the path from the attack surface to potential impact surfaces, following the data flow in the code, starting at the attack surface.
- *Command injections*: Try to abuse input fields (attack surface), and include metacharacters (such as ";" or "&") and commands in the input fields.
- *XML injections*: If the application receives XML input, try to modify that XML input to attack the parser in different ways: XML bombs, XXE attacks (to cause DoS, or disclosing sensitive data).
- *Path injections*: If the application requests a path name, use "." and ".." in the pathname you provide to try to escape any sandbox, or safe directory, or fake root directory. In Windows systems, provide as input a path name containing "/" as the separator. Inspect the code and find the name of the safe directory used, and provide a file name with exactly that name. Inspect the path from the attack surface to potential impact surfaces. To do that follow the data flow in the code, starting at the attack surface.

  Example of a vulnerability from Tapis: As a result of a command injection vulnerability, any user can execute arbitrary commands on the host where their program is being executed.

  Example of a vulnerability from Tapis: An attacker can store command injections in a Tapis database, and execute those persistent attacks when submitting a program for execution at a later time. The injection is possible because there is no validation for the name of the container image that is stored in the database when an application is created. Therefore, the container image name can include metacharacters and commands. This vulnerability is even more serious because even after it is fixed, the stored injections could still persist in the system. Only with a comprehensive scan of the system, could you be confident that you removed any residual effects of the injection attack

33.12 Buffer Overflow (or Unexpected Behaviors with Strings)

High level approach: Stress input, Experiment with the system, Focused code inspection

Through an input field, provide unexpected input, such as code, a very long input, input with metacharacters, long integers, and negative numbers.

> *Example of a bug from Open onDemand*: There were multiple issues associated with handling unusual inputs, causing unintended changes to the webpage being displayed. As a first example, a long job name resulted in a misalignment of different elements (text, buttons, and links) on the webpage, and as a consequence some elements are then inaccessible. A second example is when asking Open onDemand to create a new directory, entering invalid characters could cause unintended and bizarre directory names, such as the following name (including all metacharacters such as the quotation marks).
>
> ```
> ” title=”script>” draggable=”true”>script>
> ```

Check if the received input is sanitized. In case of server/client applications, check if inputs are only sanitized/validated on the client side. To make that check, inspect the code on the server side and see if it includes validations. Check in the code for any input that is not sanitized/validated.

> *Example of a bug from Open XDMoD:* Open XDMoD allows a user to update several fields of the portal user database, including first name, last name, email, and password. While client side validation of the database fields occurs before the requests are sent, there is no server-side validation of the first name, last name, or email fields. Since the database schema restricts the length of these fields, the lack of server-side validation forces the database to truncate the values. This truncation can lead to malformed email addresses.

33.13 Summary

FPVA is a methodology that allows an analyst to find vulnerabilities affecting the high value assets in a software system. It is a human-centric methodology consisting of five steps. This chapter covered FPVA Step 4, Component Analysis. The goal is, after understanding the big picture of the system in Steps 1–3, to find and exploit concrete vulnerabilities.

- Learn to think like an analyst.
- Understand the fourth step of FPVA, the component evaluation.
- Get a starting point for finding vulnerabilities in your system.

33.14 Exercises

1. When performing an in-depth vulnerability assessment, why should we not just start with FPVA Step 4? In other words, why are Steps 1-3 necessary?
2. (a) We consider a vulnerability to be real only when we have been able to build an exploit for it. Why should we not report on a

vulnerability we can believe is there, but for which we did not manage to construct an exploit?
(b) Why might we report on a vulnerability for which we have not (could not) construct an exploit?

3. Think about a software system with which you have experience. List what kind of specific issues, different from the ones described in his chapter, you would look for in your assessment.