Chapter 32
FPVA Step 3, Trust and Privilege Analysis

Revision 1.2, September 2025.

Objectives

- Understanding the concepts of *trust* and *privilege*.
- Relate the concepts of trust and privilege to classic operating system concepts of *authentication*, *authorization*, and *access enforcement*.
- Learn how these concepts relate to the FPVA diagrams created in Steps 1 and 2 of FPVA.
- Learn how to annotate the Step 1 and 2 diagrams to include trust and privilege information.
- Learn about tools and techniques that you can use to gather information for constructing these diagrams.

32.1 Motivation

Now that we have described the architecture and identified the key resources of the system that we are assessing, it is time to talk about how trust and privilege are handled in the system. We are concerned with two key questions: how is trust handled in the system and how are permissions given out and enforced?

Trust and privilege relate to the classic three-part operating systems model of authentication, authorization, and access control. In this chapter, we will show how trust and privilege relate to this three-part model. In fact, we will show that they are essentially different ways of breaking down the same issues.

We document information related to trust and privilege in a software system by decorating the Architecture and Resource Diagrams. Remember that we start with the Architecture Diagram and then annotate this diagram with the resources that the system uses. The diagrams that we have shown in the previous two chapters did not use colors. In this chapter, we will show how we use color to illustrate privilege information for both architectural and resource elements.

32.2 Trust

Trust boils down to the question of whether you believe what another user, computer, or process (running program) tells you or what a file or database contains. The first step in deciding whether you trust what you hear or read is whether you believe the posited identity of the source of that information. This belief is based on authenticating the identity of the other party, which falls under the topic of *identity management*. The second step is deciding whether you trust the information that comes from the other party.

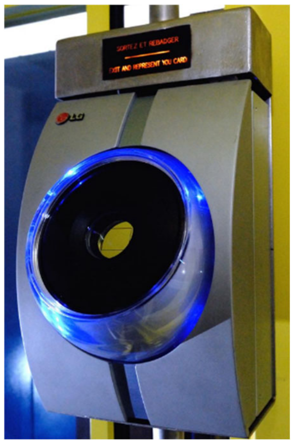
32.2.1 Authentication

Authentication refers to validating the identity provided by an entity, such as a user or a host. Authentication is crucial because if an attacker can authenticate as a valid user, the attacker will gain the privileges of that user.

Authentication might be done locally on a particular host or service, or remotely using a dedicated authentication system. The first step is for a user to convince this authentication system, either local or remote, that they are who they say they are. We usually call this the "login" step. The second step is for the authentication system to provide some sort of *token* that can be used to access services, create secure communication channels, digitally sign data, and encrypt information.

In its simplest form, login is done with a user name and password. Even with careful rules about password complexity and not reusing passwords on different sites, *password-based authentication is inherently weak and should be avoided.* The security of password-based authentication is based on keeping one fact, the password (or PIN) secret. Once this fact is exposed, your security is gone.

Alternatively, we can more types of facts, called *factors,* to enforce authentication security.



Multi-factor authentication is based on using more than one of three types of factors: what you know, what you have, and what you are. What you know means a password or PIN. What you have means having an identity card, authentication token, or authenticator application in your possession. What you are means using a biometric characteristic such as fingerprint, handprint, or retinal scan. Using a second or third factor means that simply losing a password or having it broken will not allow access to your account. Most systems today use a second factor, while the highest security facilities, such as a military installation or dangerous science facility such as the CERN Large Hadron Collider particle accelerator beam chamber.

Many systems now use identity management (IdM) systems that allow you to authenticate yourself once and then use that authentication to access many different services. Examples of such IdM systems are CILogon for science facilities or Google's single sign-on for commercial systems. IdM systems are attractive because they are typically designed by experts and will support multi-factor authentication. IdM, like cryptography, is one of those areas where you should not try to design your own system. There are many subtle ways that you can go wrong and even the smallest mistake can leave your system wide open to attack.

A common protocol used in IdM for authentication is OAuth 2.0[1], which is used to sign in to systems like Google or Microsoft. The goal of such a system is to allow you to sign on once and then use a token provided by the authentication service to access a variety of different services. Such a mechanism is often called *single sign on*. When you successfully authenticate yourself to OAuth 2.0, it returns an identification token along with an access token. Both types of tokens are typically represented as JSON Web Tokens (JWT)[2], a cryptographically signed Base64-encoded JSON object.

The Analyst's Task for Authentication

Your first task in understanding how authentication is being used in the system that you are assessing is to understand the requirements for authentication in the system. You will need this information to decide whether the mechanism chosen by the system designers is appropriate to the task. From the first two steps of FPVA, you should have a pretty good idea of what the system is doing and what kind of resources it is controlling. Certainly, if the system is controlling physical resources (a *cyber physical system*) where there is the potential for harm to people or property, then the system needs careful design with multi-factor authentication, likely three factors. If a system maintains sensitive information, including personal, financial, or medical information, then two-factor authentication is essential.

The next task is to identify what kind of authentication is being used and how it is being used. As we saw in Figure 1 in Chapter 30, a system can have more than one path of entry. For example, in that figure, we see two completely separate paths of entry. The administrator of the system accesses an administrative host through the standard operating system identity management and then using remote login (ssh) to access the system itself. This arrangement requires careful evaluation of the login procedure on the

[1] `https://oauth.net/2/`

[2] `https://datatracker.ietf.org/doc/html/rfc7519`

administrative host and then the remote login procedure. It is safe to say that in any modern system, having only password-based authentication at either of these two steps is insufficient. If the administrative host has physical controls to access it – for example, having to use a badge to access the room where that host resides – then that physical control could be considered a second factor.

Normal users have access to that system through a web interface, requiring an evaluation of the web authentication mechanism and session management. Remember that you read about web security in Chapters 20-24.

Our Economy of Design principle would argue against a design such as the one that this system has, preferring a single authentication mechanism that supports both regular and administrative users.

32.2.2 Trusting Authentication Information

The best authentication mechanism is only secure if it is based on secure communication channels. This means that all communication sent during the authentication process (and afterwards) must be encrypted. For example, any web-based operations must be based on HTTPS, remote shell operations should use ssh, and other secure communication should be based on encrypted channels using well established encryption and communication libraries like TLS[3] (formerly SSL).

Often, when information is shared, some form of secure hash is used to detect if the data has been tampered with. It is used to detect such tampering in cases such as when data is sent over a communication channel or stored in a file. Note that a secure hash protects the integrity of the data but not the confidentiality. The original data is always sent or stored with its hash value. The reader or recipient of the data can recalculate the hash of the data to see if it matches the sent or stored value. If they do not match, then tampering has occurred.

A secure hash is a cryptographic function that should have several properties, including:

1. They should be efficient in that they should not consume a lot of CPU cycles.
2. They should be one way functions, meaning that it should be extremely difficult to obtain the original text from the hash value.

[3] `https://en.wikipedia.org/wiki/Transport_Layer_Security`

3. They must have collision resistance, meaning that it should be extremely unlikely to find two different texts that would hash to the same value.

Typically, hashes from the SHA-2 and SHA-3 family are considered to be strong against attack. For example, Bitcoin cryptocurrency is protected by the SHA-256 variant of SHA-2 and Ethereum cryptocurrency is protected by the Keccak-256 variant of SHA-3.

The Analyst's Task for Trusting Authentication Information

The basic task for insuring that we can trust authentication information is to identify all data in motion, that is communication operations such as web requests and messages over sockets, and ensure that they are using encrypted channels. You will have identified these internal interactions in Step 2. In this step, you will be evaluating if that data is sent securely. Data at rest and data sent between untrusting systems should be signed with a secure hash.

32.3 Privilege

Privilege describes what each process in the system can do and what privilege is needed to access resources and external services.

Once we have determined the user ID for each running process and the ownership of each resource such as a file, then we can then annotate the Architecture and Resource Diagrams with colors that illustrate this information.

Using the user ID and ownership information, we will then evaluate authorization, which describes what each process is allowed to do, and access enforcement, which is the mechanism used to ensure that these limitations are enforced.

32.3.1 User IDs and File Ownership

A running program (i.e., a *process*) has a user ID associated with it. This user ID will determine the types of access permissions that the process has. We are particularly interested in whether this process is running with the ID of:

- A standard user, so probably relatively low privilege.
- A system user, such as a database administrator, so the process might have full access to the resource which it controls.
- The root or administrator user, so the process likely has access to any other process and all resources on that host. Since root or administrator processes have extensive privileges, they are always considered high value assets and become an early focus of the Step 4 code inspection step of FPVA.

This characteristic is true for all operating systems, including Windows, Linux, and MacOS. Every resource, such as a file, has an owner that is the user ID of the process that created the resource.

The Analyst's Task for Determining User IDs

Your first task is to determine the user ID of each running process in the system that you are assessing. As we did for Steps 1 and 2 of FPVA, we start with a running version of the system and use standard operating system commands such as `ps` on Linux and MacOS and `tasklist` on Windows. With the right options, these commands can provide quite detailed information about the running processes. We can also get a real time list of running processes with the `top` command on Linux and MacOS and the Task Manager on Windows.

To make life more complicated, processes on Linux and MacOS can change their user ID while they are running. A common scenario where this happens is when a privileged process – one that is running as "root" – wants to create a process running at a lower privilege level, which usually means on that is not running at "root". The new process is created when the existing ("root") process executes a `fork` or `clone` system call. This new process will also be running with user ID "root". To change its user ID, the new process will execute a `setuid` (set user ID) command, and perhaps a `setgid` (set group ID) command.

To detect this change in user ID, you will need to trace the system calls made by the processes in the system that you are assessing. One of the easiest ways to trace these system calls is by using the `strace` command, as we discussed in Chapter 30.

Many software systems use a configuration file to control which programs get run when the system starts and when certain commands are executed. An entry in the configuration file would have the file path name for the program and might also have a user ID at which to run this program. So, finding these configuration files (which we can see in both Figure 1 and Figure 2 in Chapter 31), is important. Such configuration files are often considered high value assets in the system.

There is a commonly used UNIX standard configuration file type called `crontab`[4], which can be used to start programs running automatically at scheduled times. These `crontab` files are read by the `cron` *daemon* (an always running system process) and run with the user ID specified in the entry in the file. Here's an example of a `crontab` entry from the file `xdmod`

[4] `https://man7.org/linux/man-pages/man5/crontab.5.html`

`crontab` used by the system illustrated in Figure 1. `cron` is starting a program to run every day at minute 0, hour 1, i.e., at 1:00 am. The program is run with user ID `xdmod`.

```
# Every morning at 1:00 AM, run shredder. Then run ingestor.
0 1 * * * xdmod /usr/bin/xdmod-shredder --quiet -r resource-name
-f slurm -I /home/user/logs/logs.log && /usr/bin/xdmod-ingestor
--quiet
```

Once we have the user ID information, we can annotate the Architecture Diagram with colors that indicate under which user ID the processes are running. So, the diagrams from Chapters 30 and 31 become the diagrams with color shown in Figure 1, Figure 2, and Figure 3.

Note that in Figure 1, the "Data Import and Processing" process has two colors. This is because it can be started from two different processes, resulting in two possible user IDs. A process could also have multiple colors if it executed a `setuid` system call while it was executing.

The colors give an easy-to-understand overview of basic privileges in the system. One of the first things that an analyst or programmer might observe is which processes are running as "root" or "administrator". We typically illustrate these in red so that they stand out. We are also interested in which processes are running as resource-specific administrator accounts, as these processes will often control shared information or configuration data.

The Analyst's Task for Determining File Ownership

As we discussed in Chapter 31, the resources created and controlled by a software system can have an influence on security. For example, files that hold credentials or configuration information can be particularly critical. If these files are stored in the wrong directory or if the access rights to these are set incorrectly, then there could be unauthorized access. If a credential or configuration file could be overwritten by an unintended user, then the future behavior of the system could be dramatically changed. If a credential file could be read by an unauthorized user, then login credentials or certificates could leak, possibly attacking confidentiality, integrity, and availability.

Similarly, the authorized removal of a file might cause significant changes in the behavior of the system. For example, removing a file that contains login credential information could cause one of three common behaviors. First, it might cause the system to crash, as the missing login information could be an unexpected error. This is an attack on availability, essentially a denial of service. Second, it might not crash the system but prevent valid users from accessing it, again, attacking availability. Third, it might all any user to access the system without valid credentials, leaving the system wide open to integrity and confidentiality attacks.

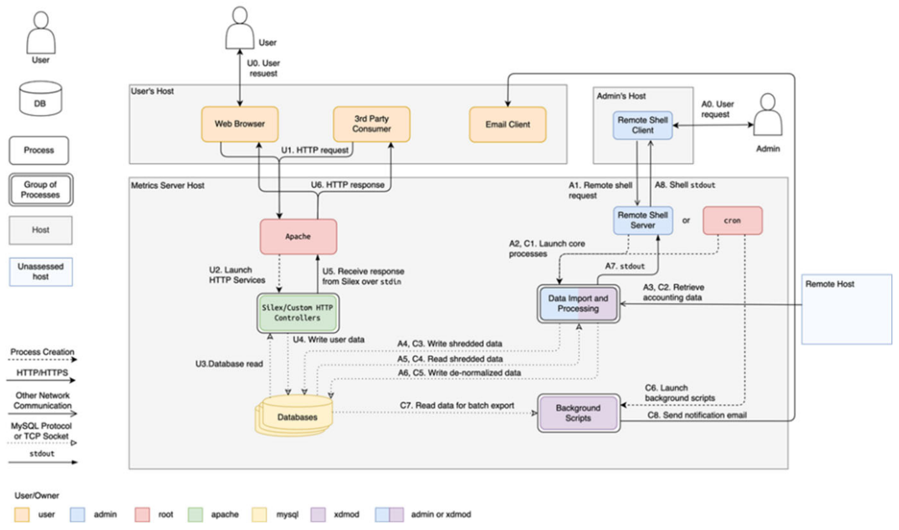


Figure 1: Architecture Diagram with User ID Information
(Based on Figure 1 from Chapter 30)

We need to check that the files are properly owned and that the access lists (file access permissions) are set correctly.

On Linux and MacOS, we need to make an extra check on executable files. As we mentioned previously, when a process on Linux or MacOS executes a program, the user ID of the new process is the same as that of the creating process. However, when the SUID (set user ID) or SGID (set group ID) permissions are set on a file, then the program will be executed with the user ID of the owner of the file. The SUID or SGID access control is sometimes called the "sticky bit". Here, we see an example of a program (`passwd`) that has the SUID bit set (shown as the "`s`" in the user permissions):

```
[2] ls -l /bin/passwd
-rwsr-xr-x. 1 root root 33544 Apr 12 2025 /bin/passwd
```

If there are only a few files in a few locations, then simple tools like the `ls` or `dir` command might suffice to review permissions and check for special cases like SUID/SGID. If there are many files, then a combination of scripts and commands might be necessary.

32.3.2 Authorization

Authorization refers to specifying access policies to resources. A system needs to enforce that a user is authorized to perform a specific operation on a resource. You need to check what privileges exist in the system you are assessing. To accomplish that, you will need to check the purpose of the different accounts associated with the system and what operations should be allowed with those accounts. For example, it is common to find administrative accounts like “root” or “admin”, and some normal user accounts. The administrative accounts will be able to affect the configuration and operations of the system, so we need to check that critical resources are accessible only by those special users.

Privileges should be fine grained. That means each privilege should limit access to the most specific operation or data item possible. This is an example of the Least Privilege design principle. Remember that this principle means that each user should be able to access only the resources they need, but no more than those resources.

32.3.3 Access Enforcement

Checking that accesses are properly enforced can take some time and detailed work. You will need to locate the parts of the code that are involved in the checking of access rights. In a well-designed system, such checking will be done in one centralized place and clearly labeled. This is an example of the Economy of Design principle.

The Analyst’s Task for Checking Access Enforcement

If a system is storing credentials and authorization, and access control information in a common database, then identifying places in the code where there are accesses to that database will help you find places where system is making these checks. If the system is using a common framework, like a web framework, then some of these checks might be integrated into that framework.

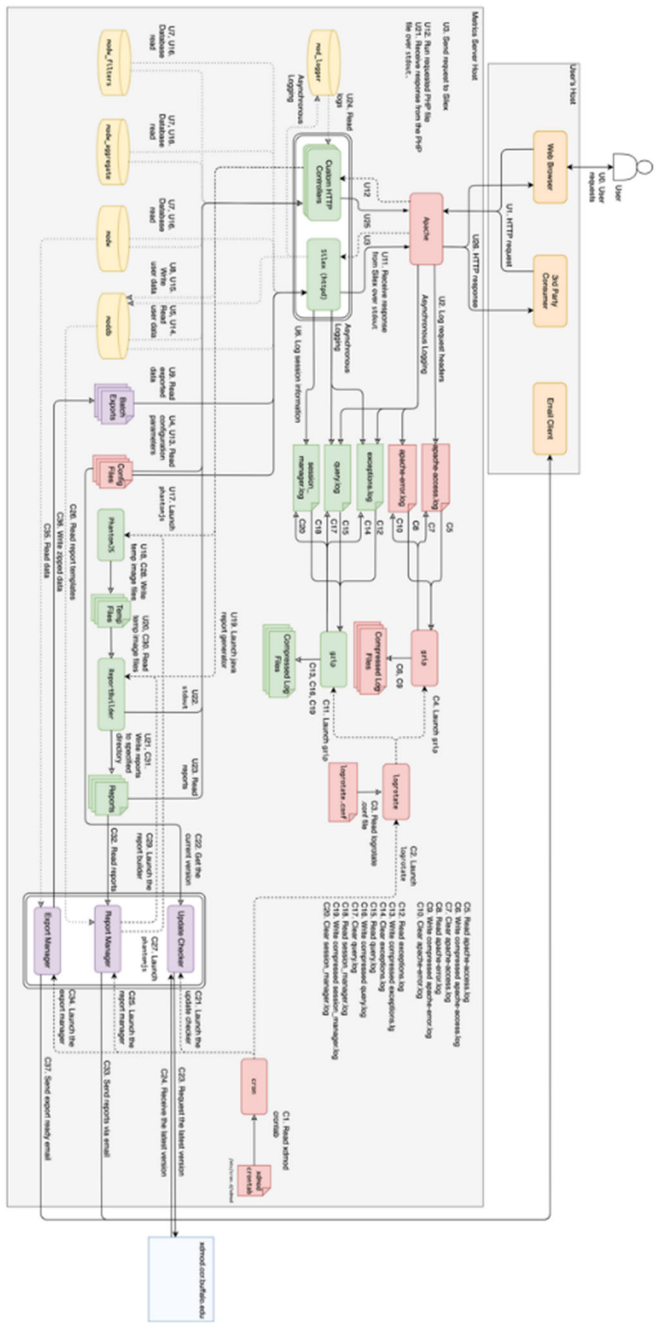


Figure 2: Architecture-Resource Diagram with User ID-Ownership Information
(Based on Figure 1 from Chapter 31)

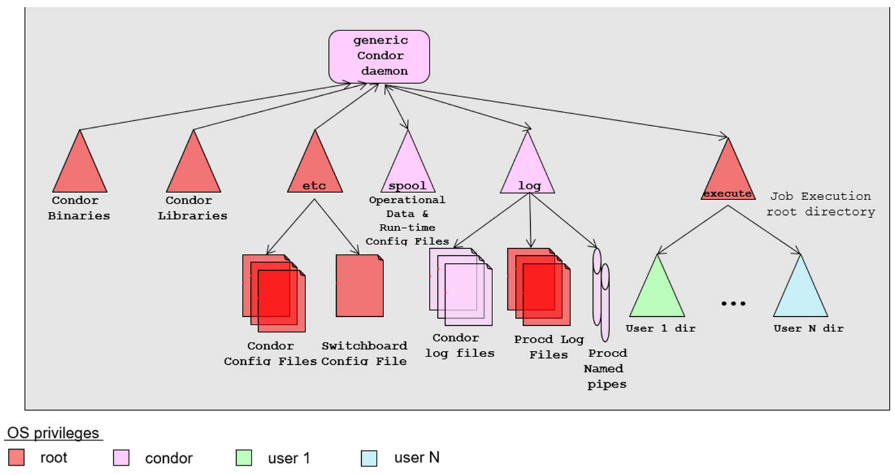


Figure 3: Resource Diagram 2 with Ownership Information
(Based on Figure 1 from Chapter 30)

32.4 Closing Thoughts

In reality, much of the information required for this step can be gathered while you are performing Steps 1 and 2, the Architectural Analysis and Resource Identification. An analyst experienced in a methodology such as FPVA will be looking ahead to this step as they are performing the earlier steps. In Chapters 30 and 31, you drew the diagrams without color and then added the color (reflecting user IDs and file ownership) in this step. Another way of thinking about this process is that you will be drawing the diagrams with color as you perform Steps 1 and 2.

Completing this step of FPVA is often a point where you start sharing your information with the team that is developing the software that you are assessing. Sharing the assessment information at this point provides a calibration for the assessment team to make sure that you are accurately understanding the system that you are assessing and running it in a way that accurately reflected real world deployments of the system. It can also provide the software development team with their first insights into your assessment activity and keep them engaged in the process.

The information gathered in this step is crucial to how the system works before you dive extensively into the code in Step 4. The diagrams and related information that you have gathered in Steps 1 through 3 will be crucial to the next step.

32.5 Summary

In this chapter, we:

- Understood the concepts of *trust* and *privilege*.
- Related the concepts of trust and privilege to classic operating system concepts of *authentication*, *authorization*, and *access enforcement*.
- Learned how these concepts relate to the FPVA diagrams created in Steps 1 and 2 of FPVA
- Learned how to annotate the Step 1 and 2 diagrams to include trust and privilege information.
- Learned about tools and techniques that you can use to gather information for constructing these diagrams.

32.6 Exercises

1. What do the colors on an Architecture Diagram and Resource Diagram represent? What should the most distinct color (like **red**) be used for? Describe how colors are used in Figure 1 and Figure 2.
2. How do the colors in an Architecture Diagram relate to those in a Resource Diagram?
3. As you did for Steps 1 and 2, for a system with which you are familiar, follow the steps in the chapter to annotate the Architecture and Resource Diagrams with trust and privilege information. You can start with the diagrams that you produced for the exercises in the previous two chapters.