Chapter 30
FPVA Step 1, Architectural Analysis

Revision 4.1, February 2025.

Objectives

- A step in learning to think like an analyst.
- Review the concept of attack surface.
- Learn how to construct an architecture diagram and the components used in an architecture diagram.
- Learn about tools and techniques you can use to gather information for constructing these diagrams.

30.1 Motivation

The goal of FPVA is to focus the analyst's attention on the vulnerabilities affecting the highest value assets. By understanding the structure of a system and the importance of its components, we can focus on the key areas of the system for detailed code analysis. Architectural Analysis is the first step in this process.

The purpose of the Architectural Analysis step is to identify the key components of the system and describe how they interact. In later steps, we will identify the resources on which the system operates (Resource Identification), how trust and privilege is handled in the system (Trust and Privilege Analysis), how to examine the code to find the actual vulnerabilities (Component Analysis), and how to disseminate the results of an FPVA engagement.

The architectural diagram is an abstraction of the software system. It captures just enough information to reason about the system's security without getting bogged down in too much detail.

We will use an architectural diagram from a real FPVA engagement as our running example. In Figure *1*, we see a diagram from a system that manages and monitors computing resources. It processes the accounting logs from a resource management system and facilitates access to the metrics through a web portal. In addition, this system can generate periodic reports and send them via email to a list of recipients. Note that this system is relatively simple so serves as a good introduction.

When drawing the architecture diagrams (or **any** technical diagram), be consistent about the use of color, shape, and position. The same feature (such as shape or color) should mean the same thing every time. Similarly, if two items have the same shape or color, then they share the same characteristic.

30.2 Starting Point: The Attack Surface

Our starting point with any system is to understand the ways in which the users (and therefore, the attackers) interact with the system. In Chapters 2 and 3, we introduced the attack surface and explained how it describes the ways that users access a system.

Remember the key lesson from these earlier chapters: if there is a path from the place where the user provides input (the attack point) to the place in the code where a serious operation is done (the impact point), and user input can affect that operation, then there is the potential for a vulnerability.

As we know, the attack surface refers to the interfaces available to the user, from user supplied data such as a web form field, network message, input file, or environment variable. If user input is not needed for an attack to succeed, then the attack is based on malware, whose mere installation causes a security issue. So, we need to look where the system gets user supplied data and understand what data users can provide to the system.

In Figure *1*, we identified two places where the users supply data. On the right-hand side we see the user admin who interacts with a remote shell client (edge A0), and on the top-left we see a regular user who interacts with the system through a web browser (edge U0). In addition, we see on the right that there is an external computer, labeled "Remote Host" that interacts with this system.

30.3 Functionality of the System

When facing a new system to assess, we need to understand the functionality of the system. Sometimes it is possible to meet with the developers to get an overview of what the system does, and how it works. Even though such an overview is helpful, you still have to check the existing documentation. Bear in mind that user manuals are more likely to be up to date than other documentation such as design documents. And even the user manuals are likely to be out of date in various ways.

To understand its functionality, you need to be able to experiment with the system. First you need to learn to use the system as a regular user, and then as an administrator. Of course, to be able to do that, you need a running version of the software. You also need access to the source code, and more importantly, access to code that can be compiled. That access allows you to modify the code to produce diagnostic and tracking information. Installing a full software stack from scratch is time consuming, so there are a couple of alternative best practices. One approach is to be provided with a virtual machine or a container with the complete software installed and running in a realistic configuration. A second approach is for the software team to provide you with remote access to a development system with the software installed and ready to run, and with the source available and ready to build.

In your first interaction with the system, you will want to learn how to run it, configure it, control it, and understand what gets logged and where.

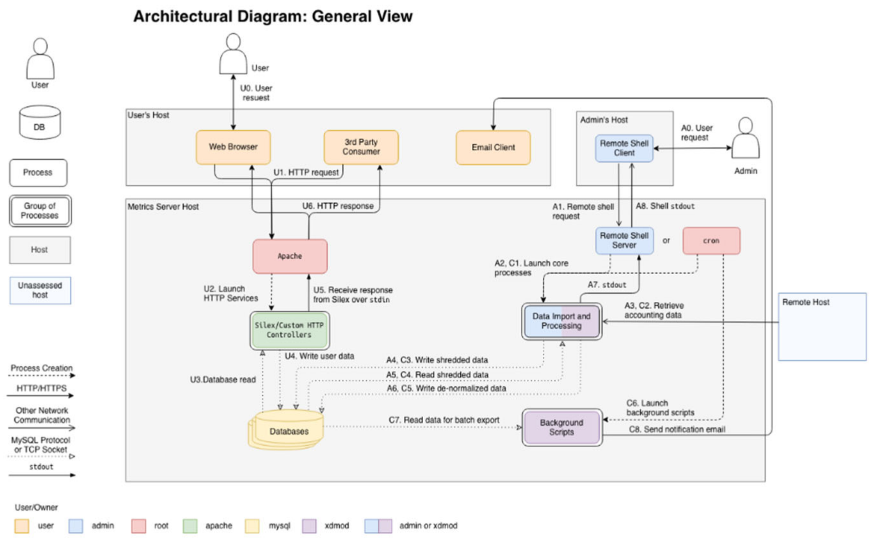


Figure 1: Example of Architecture Diagram with Attack Points

30.4 Structure of the System

Your next task is to understand the structure of the system. At this point you care about identifying the different hosts, virtual machines and containers, processes, threads, and connections (such as sockets) of the system.

30.4.1 Structure: Hosts

We start with the hosts, the computers on which the software system is running. You need to document each machine where the software was installed, or where external software used by the system is running. A host can be a server host running system executables, a client host, or both. The hosts might be physically on your premises or allocated from the cloud.

In our example system in Figure *1*, we have four hosts, each delineated by a shaded box and labeled with a descriptive name:

User's Host: This host represents the machine on which the client, i.e., the user of the system, would be running.

Admin's Host: This is another client host, distinguished from that used by a regular user. We distinguish between regular users and administrators as they have different privileges and different roles with respect to the system.

Metric Server Host: This host contains the main functionality of the system that we are evaluating. The example system is relatively simple, so there is just one host on which the system runs. In many systems, the functionality may be spread across many hosts, including things like servers or authentication, databases, file storage, execution resources.

Remote Host: does not have any processes of the system running on, but it is the host that supplies accounting data to the system that we are assessing.

30.4.2 Structure: Processes

After understanding the hosts, we now want to understand what is running on each host. The basic mechanism that an operating system provides to encapsulate a running program is called a *process*. A process contains the code, data, open files and network connections, and other attributes of a running program.

So, our goal is to understand what processes associated with the software system are running on each host. To do this step, you will need both to experiment with the system and have a look at the source code. A common starting point is to start the software system and observe the processes that are currently running on the host. Operating systems come with a variety of tools that you can use to observe the running process.

We will start with some simple command line tools. The most basic commands are `ps` on Linux or MacOS and `tasklist` on Windows. These commands list which processes are currently running at that moment (including the `ps` or `tasklist` command). Both commands have many options, so be sure to read the manual page for the command that you are using.

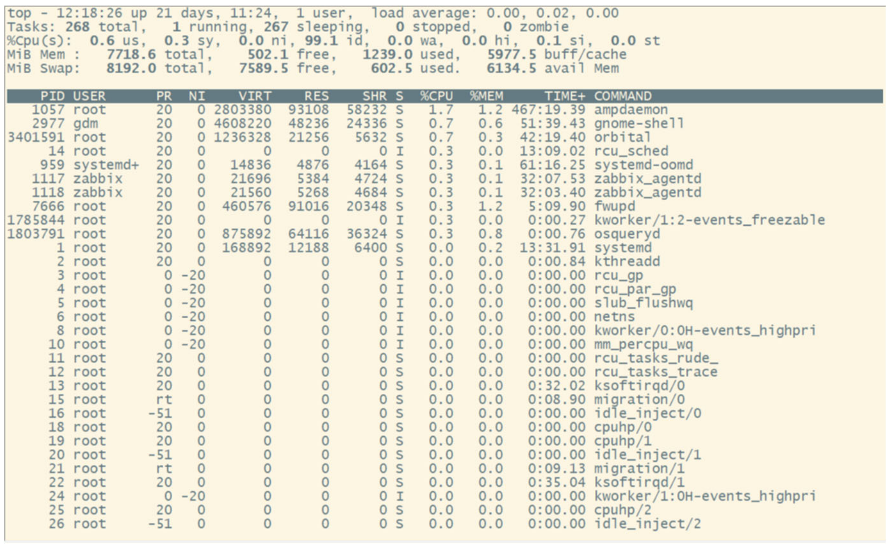
To see an ongoing and updated list of what is running on your host, you can use the `top` command on Linux or MacOS (Figure 2) or the Task Manager's process display option on Windows (Figure 3). The top command runs in a shell window and the Task Manager is run by typing CTL-ALT-DELETE and then selecting Task Manager. From there, you select the Processes option (typically by clicking on the three stripes at the top left corner of the window to show the display options). These commands not only show what is running, but also how much memory and CPU they are using and how much I/O they are performing.

For a given program, you can also run that program and trace what calls it makes to the operating system. These calls, called *system calls* or *kernel calls*, may involve I/O, networking, process control, synchronization, security functions, and many other things. For the task of identifying processes, tracing system calls can allow you to detect when a process starts up another program running (therefore creating a new process). We will also see how these commands are useful for other parts of the Architectural Analysis and later steps of FPVA.

On Linux, there is the `strace` command; on MacOS, there is the complex `dtrace` (that has its own entire D programming language), along with the much-simpler-to-use `dtruss` shell script interface to `dtrace`; and on Windows, there is the complex and sophisticated `Trace-Command`. Of these, strace is the simplest to use. The other commands require some study and use of the manual. Using `strace` to trace a program, for example a compile command, would look like:

```
% strace gcc -o myprog -g -wall myprog.c
```

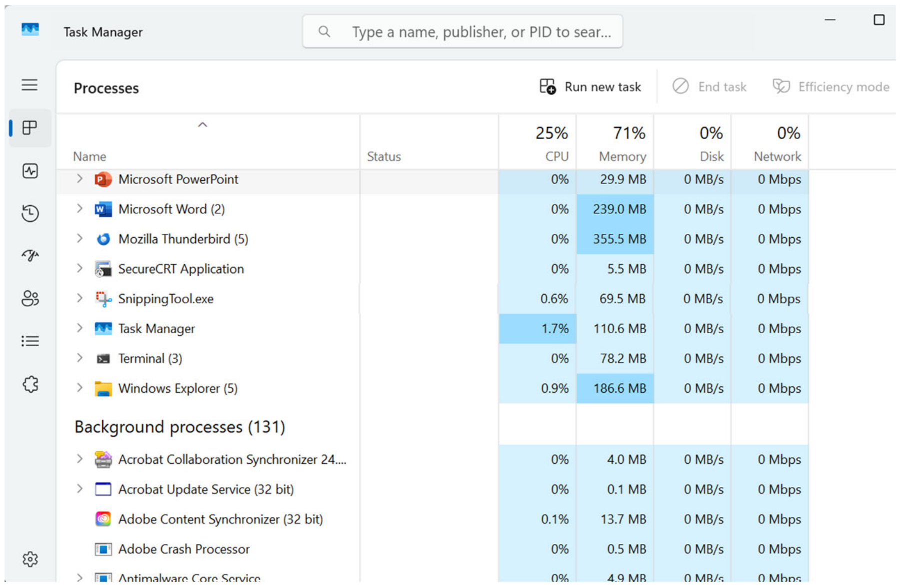
As mentioned above, using `dtrace`/`dtruss` or `Trace-Command` would require more study and the use of a reference manual.



```
top - 12:18:26 up 21 days, 11:24,  1 user,  load average: 0.00, 0.02, 0.00
Tasks: 268 total,   1 running, 267 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.6 us,  0.3 sy,  0.0 ni, 99.1 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
MiB Mem :   7718.6 total,    502.1 free,   1239.0 used,   5977.5 buff/cache
MiB Swap:   8192.0 total,   7589.5 free,    602.5 used.   6134.5 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1057 root      20   0 2803380  93108  58232 S   1.7   1.2 467:19.39 ampdaemon
   2977 gdm       20   0 4608220  48236  24336 S   0.7   0.6  51:39.43 gnome-shell
3401591 root      20   0 1236328  21256   5632 S   0.7   0.3  42:19.40 orbital
     14 root      20   0       0      0      0 I   0.3   0.0  13:09.02 rcu_sched
    959 systemd+  20   0   14836   4876   4164 S   0.3   0.1  61:16.25 systemd-oomd
   1117 zabbix    20   0   21696   5384   4724 S   0.3   0.1  32:07.53 zabbix_agentd
   1118 zabbix    20   0   21560   5268   4684 S   0.3   0.1  32:03.40 zabbix_agentd
   7666 root      20   0  460576  91016  20348 S   0.3   1.2   5:09.90 fwupd
1785844 root      20   0       0      0      0 I   0.3   0.0   0:00.27 kworker/1:2-events_freezable
1803791 root      20   0  875892  64116  36324 S   0.3   0.8   0:00.76 osqueryd
      1 root      20   0  168892  12188   6400 S   0.0   0.2  13:31.91 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.84 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_flushwq
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-events_highpri
     10 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     11 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_rude_
     12 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_tasks_trace
     13 root      20   0       0      0      0 S   0.0   0.0   0:32.02 ksoftirqd/0
     15 root      rt   0       0      0      0 S   0.0   0.0   0:08.90 migration/0
     16 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/0
     18 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     19 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/1
     20 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/1
     21 root      rt   0       0      0      0 S   0.0   0.0   0:09.13 migration/1
     22 root      20   0       0      0      0 S   0.0   0.0   0:35.04 ksoftirqd/1
     24 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/1:0H-events_highpri
     25 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/2
     26 root     -51   0       0      0      0 S   0.0   0.0   0:00.00 idle_inject/2
```

Figure 2: Sample Use of "top" Command



Name	Status	25% CPU	71% Memory	0% Disk	0% Network
Microsoft PowerPoint		0%	29.9 MB	0 MB/s	0 Mbps
Microsoft Word (2)		0%	239.0 MB	0 MB/s	0 Mbps
Mozilla Thunderbird (5)		0%	355.5 MB	0 MB/s	0 Mbps
SecureCRT Application		0%	5.5 MB	0 MB/s	0 Mbps
SnippingTool.exe		0.6%	69.5 MB	0 MB/s	0 Mbps
Task Manager		1.7%	110.6 MB	0 MB/s	0 Mbps
Terminal (3)		0%	78.2 MB	0 MB/s	0 Mbps
Windows Explorer (5)		0.9%	186.6 MB	0 MB/s	0 Mbps
Background processes (131)					
Acrobat Collaboration Synchronizer 24....		0%	4.0 MB	0 MB/s	0 Mbps
Acrobat Update Service (32 bit)		0%	0.1 MB	0 MB/s	0 Mbps
Adobe Content Synchronizer (32 bit)		0.1%	13.7 MB	0 MB/s	0 Mbps
Adobe Crash Processor		0%	0.5 MB	0 MB/s	0 Mbps
Antimalware Core Service		0%	4.9 MB	0 MB/s	0 Mbps

Figure 3: Sample Use the Windows Task Manager's Process View

As part of the architecture diagram, we show how processes are created, illustrated with dashed lines. In Figure 1, we can see process creation happening in four places. One place that process creation happens is when process “cron” launches the core process (Data Import and Processing). The edge is labeled as “C1. Launch core processes”. A second place is when “cron” later creates the Background Scripts process with edge labeled “C6. Launch background scripts”. For labeling, the analyst chose “C” to mean that the action was triggered by the “cron” daemon. The number refers to the order in which the events happen. There are two additional places where process creation happens in this diagram. See if you can find them.

30.4.3 Structure: Threads

Within a process, you can have more than one concurrent thread of execution. Each thread is running the same executable code and shares the same heap data but has its own stack. When threads are an important part of a system structure, such as for a server that runs each request in parallel in its own thread, we want to include them in our diagram.

In Figure *4* we see a portion of an architecture diagram for a system which uses threads. We can see in the BlahP process, the initial thread (labeled Server) creates a new thread to handle the execution of a new command request. To illustrate threads, we see a new arrow type (a red dashed arrow with a solid head) and circles inside of the process object to represent the threads. Knowing about threads can be important as it indicates concurrency inside the process. Along with concurrency comes the use of shared memory between the threads, the need for synchronization (such as locking), and the potential for concurrency (race condition) errors.

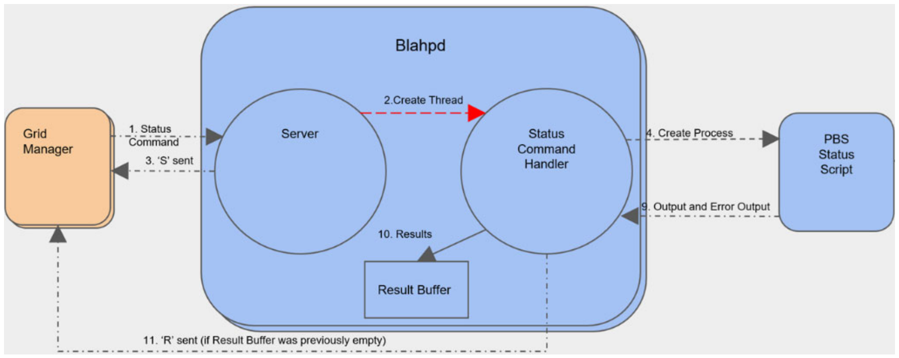


Figure 4: Architecture Diagram for System with Threads

30.4.4 Structure: Interactions and Communications

The next step in the Architectural Analysis is to understand the communication among the processes. We need to understand the communication that exists between the different processes of the application. This step can be the most challenging part of the Architectural Analysis because you have to find the communication channel creation and the communication operations. Basically, we need to discover which processes are involved in that communication, what mechanism is used for communication, what is communicated, and when (the order of communication events). *This step will provide valuable information for Step 3, Trust and Privilege analysis.*

Common forms of communication include command line arguments and environment variables when starting a program, messages over sockets, and pipes signals (on Linux and MacOS), message queues (Windows), shared memory, and memory-mapped files. In the rest of this section, we provide more detailed guidance on several of these mechanisms.

Web (HTTP/HTTPS) Communication

We can see in Figure 1 that some communication is shown by solid arrows. Web-based communication (HTTP or HTTPS) is so important and prevalent that we use a distinct arrow style for this type of communication. The User makes a request from their Web Browser (edge U0) and this request is delivered over an HTTP connection (edge U1) to the Apache web server. In response to this request, Apache launches a new process (edge U2) and receives data back from that new process (edge U5).

SQL Communication

Another form of communication in this system is with database services via some variant of the SQL protocol. Since this type of communication is also important and prevalent, a distinct type of arrow (dotted lines with hollow arrowhead) is used to denote it. In the request sequence we just described above, the process created by Apache then both reads from (edge U3) and writes to (edge U4) the Database server.

Other Network Connections

While web and database access are special (though common) cases of network communication, programs often use network connections, called *sockets*, to communicate between processes. It is essential to map out these connections so that you can use this information in Step 3 (Trust and Privilege Analysis) to understand how the system functions, how data is transmitted, and trust is communicated between processes. For example, in Figure *1*, a process on the Remote Host is communicating with the Data Import and Processing process (edge A3). Since this is an external connection, it could be of special concern for security.

Command Line Arguments

If a system can start new processes, such as with edge U2 in our example, then it likely passes parameters to the new program being started. If an attacker can manipulate these parameters, then it might be able to cause a program to do something different than you intended. As a simple example, consider the standard pattern searching program on Linux and MacOS, `grep`. If you somehow added the `-v` parameter to a `grep` command, then you would reverse the meaning of the command, i.e., you would be searching for the lines that do *not* match the pattern instead of the lines that do match.

Environment Variables

Another mechanism that communicates information to a program being started is the *environment variable*. Environmental variables are strings in the form `key=value` pairs passed to a process from its parent (the process that is starting the new program). Some environment variables have critical systems implications, like the list of directories to search for dynamically linked / shared libraries; on UNIX systems, LD_PRELOAD and LD_LIBRARY_PATH control this selection. There is also the variable that controls which directories to search when executing a new program; on UNIX systems, PATH controls this selection. Other environment variables might be application specific, perhaps having serious security implications. If there is a possible path from the attack surface to the setting of these variables, then a careful analysis of this path is essential.

Signals

On UNIX systems signals are asynchronous notifications sent to a process. A signal can come from the operating system, such as the SIGXFSZ signal sent to a process when it writes to a file that exceeds the maximum allowed size. A signal can come from runtime events, such as the SIGFPE signal sent to a process when it executes a problematic arithmetic operation, such as division by zero. Or it can come from another process, such as the SIGKILL signal sent to a process to cause it to terminate.

If an attacker can cause a signal to occur, then it can change the behavior of the program.

Shared Memory / Memory-Mapped Files

Processes can also share information with each other by means of a section of memory that they can all read and write.

A common way of sharing memory is for multiple processes to map the same file into their respective address spaces. On Windows, you would call `CreateFileMapping` followed by a call to `MapViewOfFile`. On Linux and MacOS, you would use `mmap` after opening the file. Finding such calls in a program is a sign that they use shared memory. Unfortunately, once a mapped region is created, accesses to it are done by normal pointer-based memory operations. So, while it is easy to find calls to the above functions, it can often be quite difficult to find the places in the code where the actual references to the shared memory are.

30.5 Multi-Level Diagrams

It is important to provide a version of the architecture diagram that fits on a single page. Such a diagram gives the analyst and development team an easy-to-understand overview of the entire system. And for many systems, a single page is sufficient to provide reasonable detail of the system. However, there are cases where one page is not enough. In those cases, we recommend creating a one-page version of the diagram that hides some of the details. Then provide ancillary diagrams that provide a more detailed view of some parts of the system.

For example, in Figure *5* we see an architectural diagram from a real assessment. It has many of the same characteristics as the diagram in Figure 1, however the analyst wanted to illustrate the important software packages that were being used in the complex Flash Runtime process. To accomplish this goal, the analyst created a "zoomed" version of the diagram (Figure 6) to show these packages. They also included a legend that introduced the new object shape for packages.

30.6 Closing Thoughts

With a well-constructed architecture diagram, you are ready to identify the resources that the system is accessing.

We note that there is no fixed formula for building architectural diagrams. As systems, programming languages, and computer architectures evolve, we find new mechanisms that need to be illustrated in an architecture diagram. The design, functionality, and complexity of the system that you are assessing will shape the diagrams that you will produce.

30.7 Summary

In this chapter, we:

- Made a step forward in learning to think like an analyst.
- Reviewed the concept of attack surface and showed how to reflect that in an architecture diagram.
- Learned how to construct an architectural diagram and the components used in an architectural diagram.
- Learned about tools and techniques you can use to gather information for constructing these diagrams.

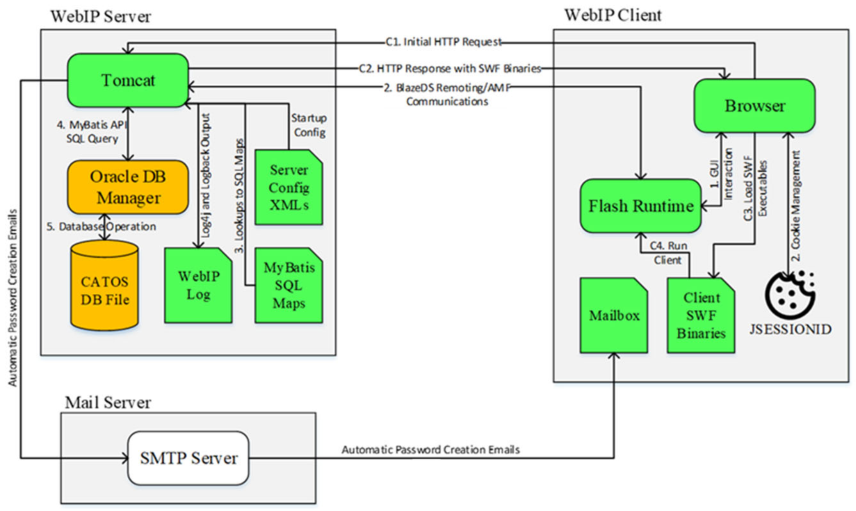


Figure 5: Architecture Diagram 2, Unzoomed

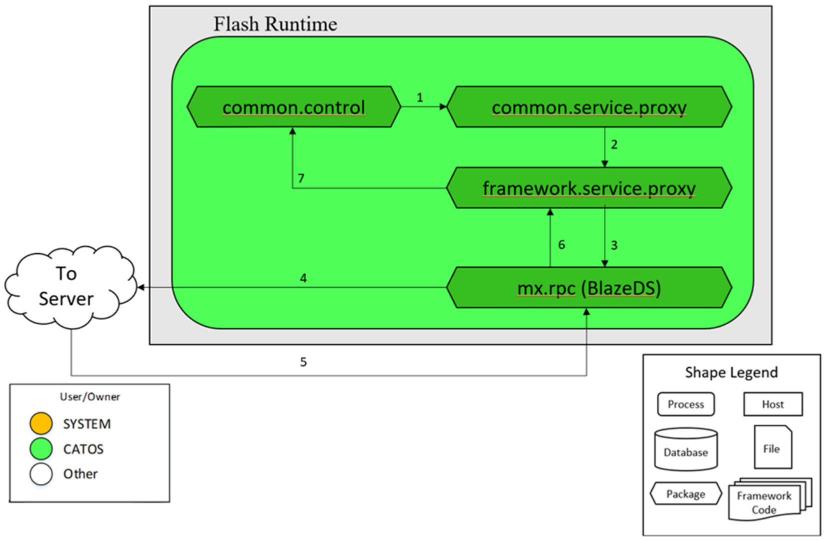


Figure 6: Architecture Diagram 2, Zoomed in Flash Runtime Process

30.8 Exercises

1. Experiment with a variety of tools for listing the processes running on your computer.
2. In Figure 1, there are two places where process `cron` creates other processes. There are two additional places where process creation occurs in this figure. Identify these places.
3. For socket-based communications:
   a. Find as many operating system kernel/system calls as you can that are involved in this type of communication. Of course, the essential one is `socket`, however there are many others. Hint: `connect` and `accept` are two more such calls.
   b. Explain the meaning of each call that you found.
   c. In interpreted languages such as Java, Ruby, or Python, these calls are located in special libraries or packages. For your favorite interpreted language, find out how to access these functions.
4. For a system with which you are familiar, follow the steps in the chapter to construct an architecture diagram for that system.
5. If you are familiar with Microsoft Threat Modeling (Chapter 5), compare how a threat modeling diagram compares in structure and context to an FPVA architectural diagram.