

Runtime Enforcement with Behaviour Control

B.Tech Project Report

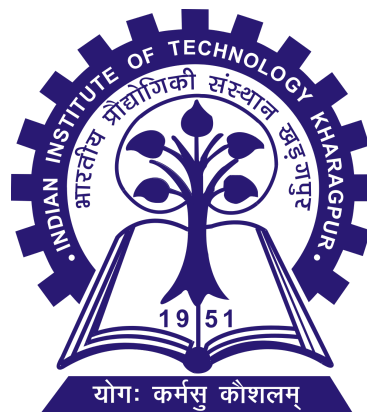
Submitted By

Shivam Saxena

17EC10054

Under the esteemed guidance of

Prof. Manoj K. Mondal



Rajendra Mishra School of Engineering Entrepreneurship

Indian Institute of Technology, Kharagpur

CERTIFICATE

This is to certify that the project report entitled “**Runtime Enforcement with Behaviour Control**” is submitted by Shivam Saxena (17EC10054) to Rajendra Mishra School of Engineering Entrepreneurship, in partial fulfilment for the award of B.Tech degree, is an authentic record of the BTP Project-2 carried out by him under my supervision and guidance. The report has fulfilled all the requirements as per the regulations of this institute.

Date: May 6, 2021

Prof. Manoj K. Mondal
Rajendra Mishra School of Engineering Entrepreneurship
Indian Institute of Technology Kharagpur

DECLARATION

I hereby declare that the project entitled “**Runtime Enforcement with Behaviour Control**” submitted to the Rajendra Mishra School of Engineering Entrepreneurship, IIT Kharagpur, in partial fulfillment for the award of B.Tech degree is the BTP Project-II work done by me under the guidance of Prof. Manoj K. Mondal.

Date: 6th May 2021

Shivam Saxena

Electronics and Electrical Communication Engineering

Indian Institute of Technology Kharagpur

Table of Contents

| | |
|------------------------------------|-----------|
| Table of Contents | 4 |
| Introduction | 5 |
| Application Modelling | 5 |
| Motivation | 5 |
| Types of Transaction Models | 6 |
| Scalability with Transaction Model | 7 |
| Discovery | 8 |
| Runtime Enforcement | 9 |
| Policy Object Model | 9 |
| Execution Enforcement | 14 |
| Remote Code Execution | 14 |
| Data Access Enforcement | 15 |
| SQL injection | 17 |
| References | 19 |

Introduction

Application security is the process of making apps more secure by finding, fixing, and enhancing the security of apps. Much of this happens during the development phase, but it includes tools and methods to protect apps once they are deployed. With more businesses and services moving their operations to app-based environments, it is becoming very important to develop ways to thwart malicious attacks by hackers.

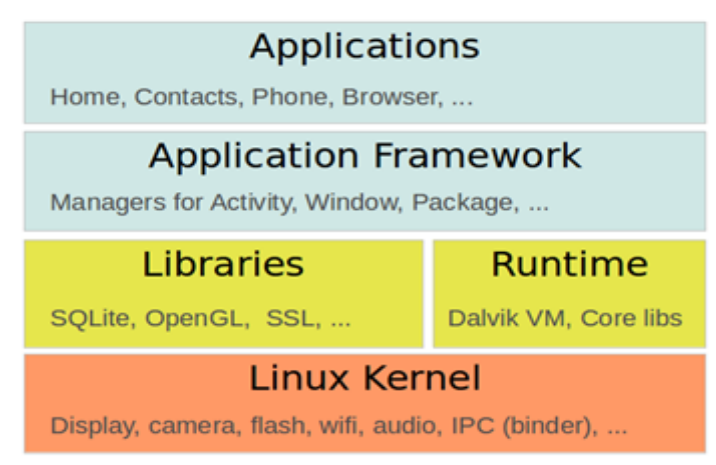
An important aspect of application security is runtime security, whereby we monitor applications and generate alerts at runtime itself. To tackle this problem, we first need an understanding of the various execution models used by modern applications. Using this, we intercept the execution patterns at the syscall level, and influence their behaviour under certain conditions defined in our controls. This enables us to halt malicious software and allow smooth functioning of the application's processes. We call this modelling and enforcement of policies as behaviour control.

Application Modelling

Motivation

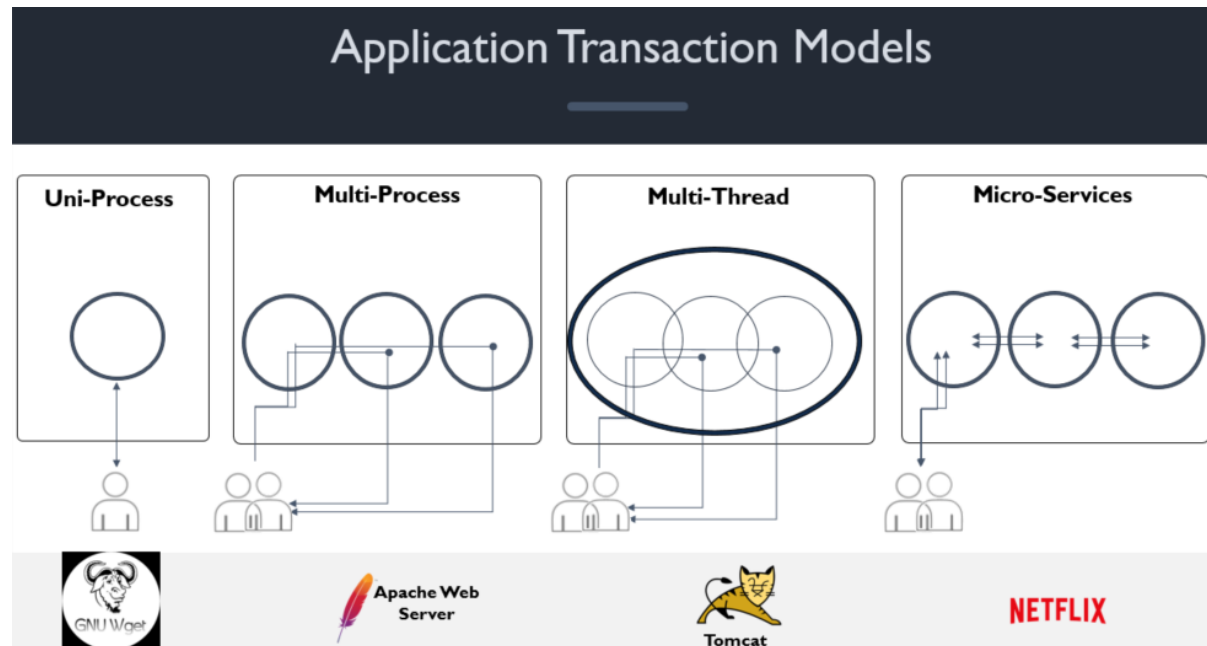
The majority of applications have a well-defined set of APIs that export various services to their users. These applications are created using a layered software stack that includes custom code, middleware, and other components (with various libraries). When clients use these APIs, the application processes are triggered to run code from middleware libraries and custom code (business logic layer)

The transaction model of an application describes how the application's various threads communicate and execute code to provide the service that corresponds to different APIs. Some of these threads are ephemeral, meaning they are created for a specific transaction and then terminate once the response is received.

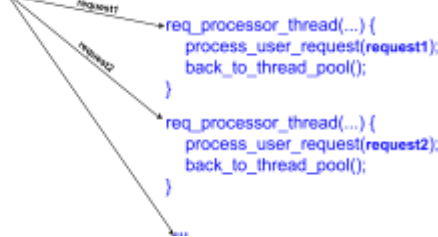



Types of Transaction Models

Following is a list of the types of transaction models with example commercial applications.



| | Transaction Model Type | Example | Pseudo-code |
|---|------------------------|---------------|---|
| 1 | Uni-Process | Wget-1.13 | <p>Each end-user request is handled one at-a-time by the single application process that executes all the application code.</p> <pre> // single process // serve one user request at a time main_server_loop(...) { new_socket = accept(...); // get new user request handle_request(); // handle request } // forever </pre> <p>The application is ready to service requests so long as the process is alive.</p> |
| 2 | Multi-Process | Apache-2.2.21 | <p>The application process creates a new process to handle (upto a max concurrence limit) user requests and execute all the application code.</p> <pre> // multiple processes // each process serves a different user at a time main_server_loop(...) { new_socket = accept(...); // new user if ((pid=fork()) == 0) { // child process execve(...); // handle request } // parent process goes back to loop } // forever </pre> <p>The newly forked processes are ephemeral in that they terminate after the transaction is serviced.</p> |
| 3 | Multi-Thread | Tomcat-7.1 | <p>Each end-user request is handled one at-a-time by a pool of threads (upto a max concurrence limit) each of</p> |

| | | | |
|---|------------------------------|---------|---|
| | | | <p>which executes all the application code.</p> <pre>// pool of request processor threads // each thread serves a different user request at a time main_server_loop(...) { ... // 15 threads to handle 15 requests at a time start_req_processor_threads(15); ... }</pre>  <pre>req_processor_thread(...) { process_user_request(request1); back_to_thread_pool(); } req_processor_thread(...) { process_user_request(request2); back_to_thread_pool(); } ... </pre> <p>The actual work of processing the request is handled by a newly created ephemeral thread (or multiple ephemeral threads) that terminates once the request is serviced.</p> |
| 4 | Event-Driven / Microservices | Netflix | <p>Application is composed of a set of services running as separate processes. Each end-user request is handled by a chained series of these services based on the type of the request. Upon service1 completion event, service2_thread() picks up and so on. The transaction is thus handled asynchronously by a series of microservices.</p> <pre>// Event Driven Architecture - Microservices service1_thread(...) { process_user_request(request1); back_to_thread_pool(); } service2_thread(...) { process_user_request(request1); back_to_thread_pool(); } ... serviceN_thread(...) { process_user_request(request1); back_to_thread_pool(); } </pre>  <p>In this architecture, the threads shown are all persistent during the lifetime of the service/application.</p> |

Scalability with Transaction Model

Every application transaction's runtime behaviour is determined by the Transaction Model. The following are some important considerations for any application's transaction model.

1. As previously mentioned, there are four different types of transaction models. In most server-side applications, the multi-threaded and event-driven (microservices) models are used.
2. The transaction model is invariant to factors such as,
 - a. scale (invariant due to no. of concurrent transactions, CPUs, memory, etc.),
 - b. the platform (bare-metal, virtual machine, cloud, container, etc.)
3. The transaction model is determined by the application server (Eg. Tomcat, Oracle Weblogic, MS IIS) including the frameworks (Struts, Ruby, etc.), and runtime (Java JDK). The application server contains "middleware" with various other layers of the software stack including "framework" (Eg. Struts, .NET, Spring), the language runtime (Java JDK).

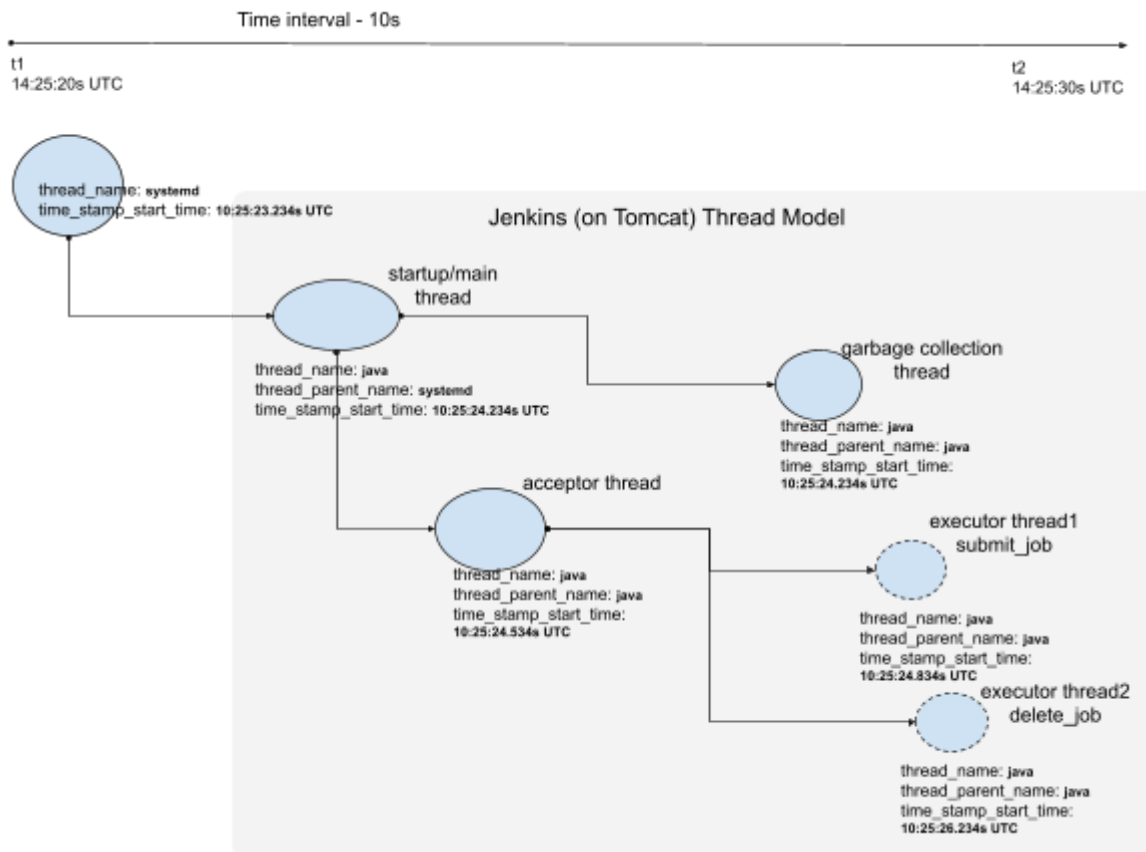
The business logic part (the top layer, "custom code", of the Application stack), uses the transaction model as defined by the application server.

The underlying application servers (middleware) that are used in software are much smaller, despite the fact that the applications are complex and designed to do various things. Around 65 percent of all Java programs, for example, use the Tomcat Application server.

Consider the Jenkins application, which runs in a Tomcat application server and is a common Enterprise automation server for source code build and release management.

The system initialization service "systemd" starts the Tomcat server's main thread, which then starts several other threads with specific features, as shown in the diagram below. The acceptor thread, for example, is in charge of accepting all client link requests. A pool of executor threads is also started by the main thread (based on configuration).

One or more processes, each with one or more threads, make up the application. The parts that follow are focused on the Linux operating system. The relationship is similar for other operating systems (e.g., Windows, Solaris, etc.).



Discovery

Each of these transactions has the following phases,

1. "acceptor" - accept a particular client request
2. "executer" - start processing the request (API specific operations)
3. "data access" - connect with mysql database to act upon data relevant to this transaction
4. "response" - finally send response back to the client

For each of the application APIs, the runtime normal behavior consists of,

1. Execution Pattern
Syscalls executed in each phase of the transaction by a certain set of threads in a sequence.
2. Data Access Pattern
Data that is accessed within the context of a particular transaction. This includes any local data (local files, etc.) or remote data (filesystems, databases).

We can model all of the application APIs if we monitor the syscalls on a machine.

However, there are two difficulties in putting such a tracking system in place. The first is that even simple tasks performed by the program require thousands of syscalls. As a result, we'll need a way to collect and condense useful data from these raw syscalls. The other challenge is all these syscalls look different on different systems, so we need a way to standardise them. These issues can be resolved with various normalization and summarization mechanisms.

With this information at our disposal, we delve into our runtime enforcement methodology. A kernel module was developed to take care of all steps in our mechanism.

Runtime Enforcement

Policy Object Model

One or more deviant application transactions of a certain nature are specified using a json format. Using this object as input, the “prevention” engine enforces the rules specified by the input to block the deviant behavior of the application on the host during the runtime.

The result is to prevent breaches such as data exfiltration, unauthorized access etc. The policy object model used for this purpose is as follows:

Example Input

```
{
  "controls" :
  [
    {
      "eventId" : "sys_execve",
      "action" : "FAIL",
      "applicationName" : ["test"],
      "flines" :
      [
        { "tName" : "java", "pName" : , "tId" : , "pId" :
          },
        {
          }
        ]
      }
  ]
}
```

The above json used for passing controls to the kernel has following attributes:

- controls : This is a *json-list* of *json-object* that describes one control.

The attributes of the *json-object* are described below.

- eventId : It is a string storing the name of the function whose execution path is to be changed.
- action : It contains one of the following values which determines how the control is enforced.
 - TERMINATE : Kill the stated process if the process is observed executing this function.
 - FAIL : Fail the function call for the given call.
 - ALLOW : Allows the function call for given call(default)
- applicationName : The application for which this control is to be enforced.
- fline: It is a list of objects containing the lineage of the thread in the form of a *json-list* which is supposed to be affected. Each next entry describes the next thread in lineage.

This list contains *json-objects* which each describes a thread by the following attributes :

- tName : Name of the thread.
- tId : Id of the thread.
- pName : Name of the process associated with this thread.
- pId : Id of the process associated with this thread.

The kernel module developed supports an IOCTL interface to get this information from the userspace and enforce it on the host. An *ioctl*, which means "input-output control" is a kind of device-specific system call. There are only a few system calls in Linux (300-400), which are not enough to express all the unique functions devices may have. So a driver can define an *ioctl* which allows a userspace application to send it orders.

The **ioctl()** system call manipulates the underlying device parameters of special files. Many core features of character special files (e.g., terminals) can be managed using *ioctl()* requests.

It is defined in the header file `sys/ioctl.h` and its syntax is as follows :

```
int ioctl(int fd, unsigned long request, ...);
```

- The argument *fd* must be an open file descriptor.
- The second argument is a device-dependent request code.
- The third argument is an untyped pointer to memory. It's traditionally **char *argp** or **void *argp**

An **ioctl()** *request* has encoded in it whether the argument is an *in* parameter or *out* parameter, and the size of the argument *argp* in bytes.

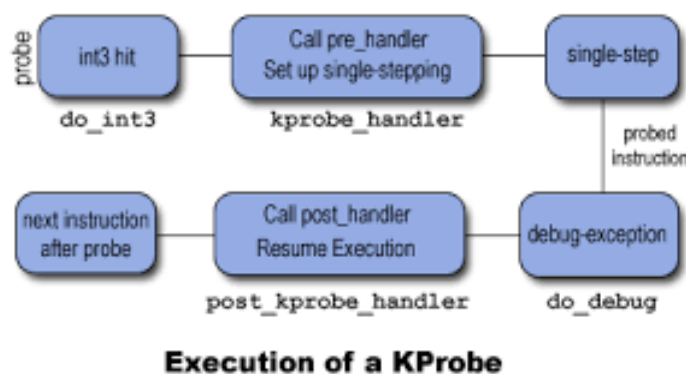
Influencing Execution Path

Execution path of the program is affected by making the requested system-call fail in the middle of the execution. This is accomplished by changing the values of the function arguments to a value which makes sure that the function would fail.

This is done using linux kretprobe functionality. kretprobe allows us to call a user defined function before starting the syscall execution and after the system call returns. Kretprobes are implemented in the linux kernel using kprobe.

The following is the flow execution:

- User program calls the syscall.
- Entry handler of the kretprobe is invoked.
- syscall is executed.
- Return handler is called.
- Control returns to the user program.



Looking at the function we want to fail e.g execve-

execve takes 3 arguments - name of the file to be executed, list of command line arguments and environment variables. execve can be failed by changing the first argument to 0 (which is stored in the rdi register in x86_64 architecture). This is done in the entry_handler of kretprobe.

When the syscall returns the handler of kretprobe is called, here we can change the value of rax (register storing the return value of the syscall) to set the error number to whatever we desire.

The entry_handler and handler function pointers which are the member of struct kretprobe. To set a function as handler or entry_handler just assign the functions to these pointers.

Sending Controls to Kernel Module

Structures

As discussed above, controls are defined in a json file which is being sent to the kernel using an ioctl interface. The kernel module parses the send json file and stores the process information in struct Control.

The control structure contains information about the control, the function whose execution by mentioned thread is to be altered and information about the thread for which this control should be enforced.

```
struct Control{
    char functionName[NAME_MAX];
    int returnValue;

    struct kretprobe rp;
    struct ThreadInfo * threadInfo;
    int lineageCount;

    unsigned long key;
    struct list_head node ;
};
```

- **functionName** : name of the function to be probed using the kretprobe.
- **returnValue** : new return value of the probed function.
- **rp** : instance of kretprobe that will be probing the specified function (documentation).
- **threadinfo** : pointer to a struct that stores information about the thread and its lineage.
- **lineageCount** : count of how back are we going in the lineage of the thread while checking for ancestry. For above json it would be 2 in both cases.
- **key** : key is the value by which the control is searched. By default it is 0 but when the control is enforced (kretprobe is inserted) the value of key is set to value of rp typecast to unsigned long.
- **node** : instance of struct list_head which makes struct Control instances, part of a linked-list.

(NAME_MAX is defined in limits.h and has a value of 255)

Each instance of struct Control has a pointer to struct ThreadInfo member which stores the information about the thread associated with the control. struct ThreadInfo stores information about the thread lineage and the parent process in form of NULL terminated linked list. Having the head at the information about the task and going back to the last generation with every Node->next. Implementation of linked list is present in the List.h.

The ThreadInfo structure contains information about the thread (its name, its lineage).

```
struct ThreadInfo{
    char threadName[NAME_MAX];
    char processName[NAME_MAX];
```

```

    int threadId;

    int processId;


    struct ThreadInfo * parentInfo;

};

```

- threadName : name of the thread defined by this instance of struct ThreadInfo.
- processName : name of the parent process of the thread defined by this instance of struct ThreadInfo.
- threadId : id of the thread defined by this instance of struct ThreadInfo.
- processId : id of parent of the thread defined by this instance of struct ThreadInfo.
- parentInfo : pointer pointing to the struct ThreadInfo instance which stores the information about the thread who comes above the thread defined by this instance of struct ThreadInfo in thread lineage (NULL if not specified).

Along with ThreadInfo the Control also has kretprobe as a member. This kretprobe is the one associated with this particular instance of struct Control and enforces this particular control.

The module creates a char device /dev/preventionDevice and the user can perform ioctl operations on this file to send the controls to the module via a json file.

Control Flow

Adding Controls to Prevention Module

The user program calls ioctl with PREVENTION_LOADFILE command and sends in a struct FileInfo instance which contains name(full path name) and size(bytes) of the json file. PREVENTION_LOADFILE and other ioctl flags are defined in the prevention.h.

A json file can also be added while loading the module by setting mode="ENABLE", json=<full_path> and size=<number>. The path is of type string thus it needs to be in double quotes. If the mode parameter is set to "ENABLE", json and size are necessary to be set. If not done the module insertion will fail. Module insertion would also fail if the file passed has invalid json.

Checking for Valid Thread

Because the kretprobe are implemented in linux using kprobe for each function call to the registered function (the function for which the kretprobe is inserted) the entryHandler and retHandler will be called. (Here entryHandler and retHandler are the functions that are assigned to the entry_handler and handler function pointers)

Thus to check that the execution path for the calling thread should be changed or not the entryHandler calls validateTask function which checks the thread attributes and thread lineage returns 0 if the thread is the one mentioned in the rule.

If the thread is valid then entryHandler and retHandler makes appropriate changes to the values of the registers thus influencing the execution path.

Enforcing Controls

We make use of a hash table to store and retrieve controls. A hash table is a key-value data structure that allows common operations to be executed in constant time. The control and its associated kretprobe act as the unique key for hashing. The controls are stored in a list where they are processed one after another and if the control is valid, appropriate policies are enforced.

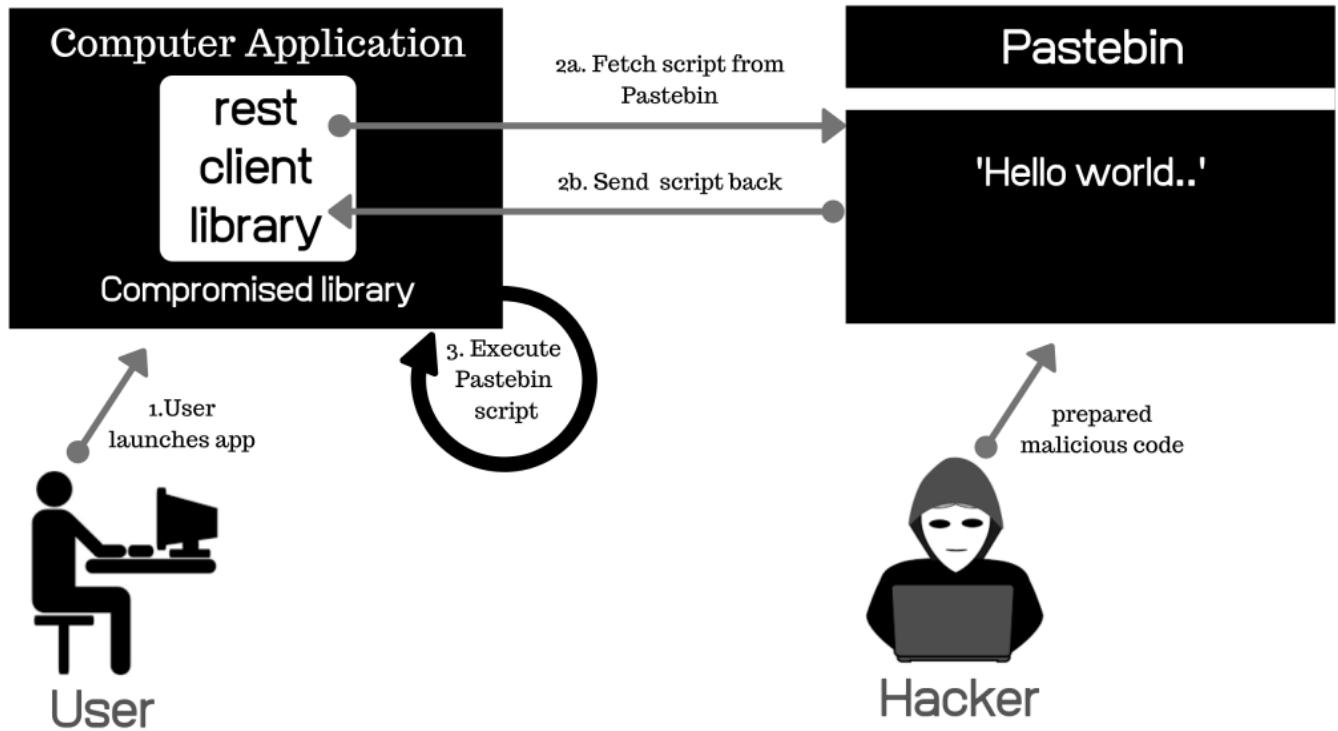
Now we take a look at two broad areas of application of behaviour control, namely execution enforcement and data access enforcement.

Execution Enforcement

To prevent malicious code running in the background that may cause serious damage to a user's system, we implement execution enforcement. A lot of programs on Linux are internally executed via the exec() family of syscalls, chief among them being execve(). By preventing unauthorized calls to this function, we can keep a strong check on the kinds of programs running on a given system. We explore this area in the context of remote code execution.

Remote Code Execution

Remote code execution (RCE) is a class of software security flaws. An intruder can use this vulnerability to run code with system level privileges on a server that has the requisite flaw. Once a server has been sufficiently corrupted, an intruder may be able to access any and all information on it, including databases containing information provided by oblivious clients. RCE belongs to the broader class of arbitrary code execution (ACE) vulnerabilities. With the internet becoming ubiquitous RCE vulnerabilities' impact grows rapidly.



To stop RCE attacks, we generate a control as follows :

```

{
  "controls" :
  [
    {
      "eventId" : "sys_execve",
      "action" : "FAIL",
      "applicationName" : ["test"],
      "fline" :
      [
        { "tName" : "java", "pName" : "java", "tId" : "2245", "pId" : "2243" },
        { "tName" : "java", "pName" : "java", "tId" : "2245", "pId" : "2243" }
      ]
    }
  ]
}

```

This control specifies that :

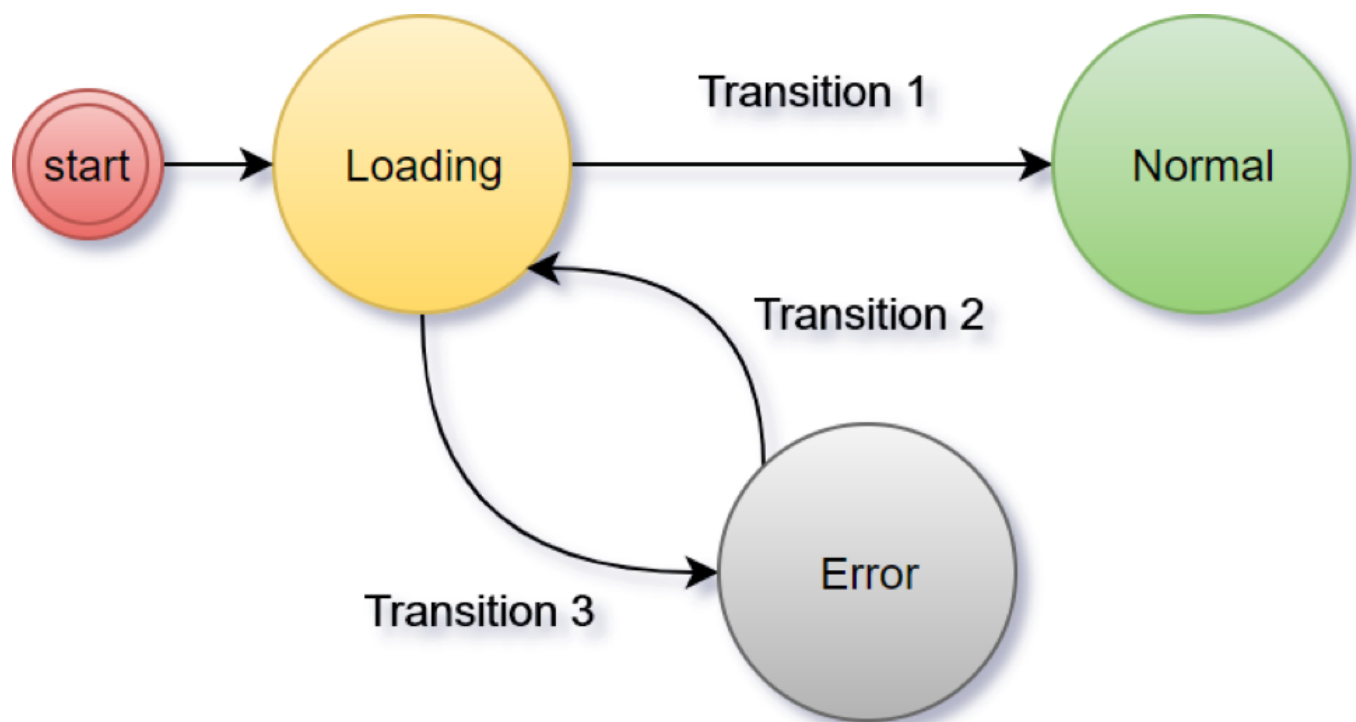
1. Monitor the application 'test'
2. For a thread with identifiers (tName, pName, tId, pId) specified above:
 - a. Intercept all sys_execve calls
 - b. Fail their execution

Using this type of control gives us sufficient information to discover the threads in question and modify their execution. Extending this to other exec syscalls can provide us with a wider array of pathways to stop malicious programs.

Data Access Enforcement

The above methodology, though powerful, lacks an important aspect: how to monitor several syscalls in succession. It may be the case that a certain syscall needs to be failed depending on the sequence of events that led up to it i.e the state of the process. This leads to an obvious extension of the above approach: a finite state machine.

A finite state machine (FSM) is a mathematical abstraction used to design algorithms. A state machine is designed to read a set of inputs. It will turn to a different state when it receives an input. For a given input, each state determines the subsequent state to move to. At any given moment, the FSM can be in one of a finite number of *states*. In response to certain inputs, the FSM will shift from one state to another; this is referred to as a *transition*.



The state machine is used to model the various steps involved when working with files. For e.g when a text file is to be modified the following syscalls are executed:

1. `open()`, to open the given file, in a given mode (read-only, read-write etc).
2. `read()`, to read the contents of the file.
3. `write()`, to add/delete content in the file.

4. close(), to close the given file

So we have different states with respect to a given file, and execution of the syscalls creates transitions to other states.

To accommodate the state machine, we update our control json as follows :

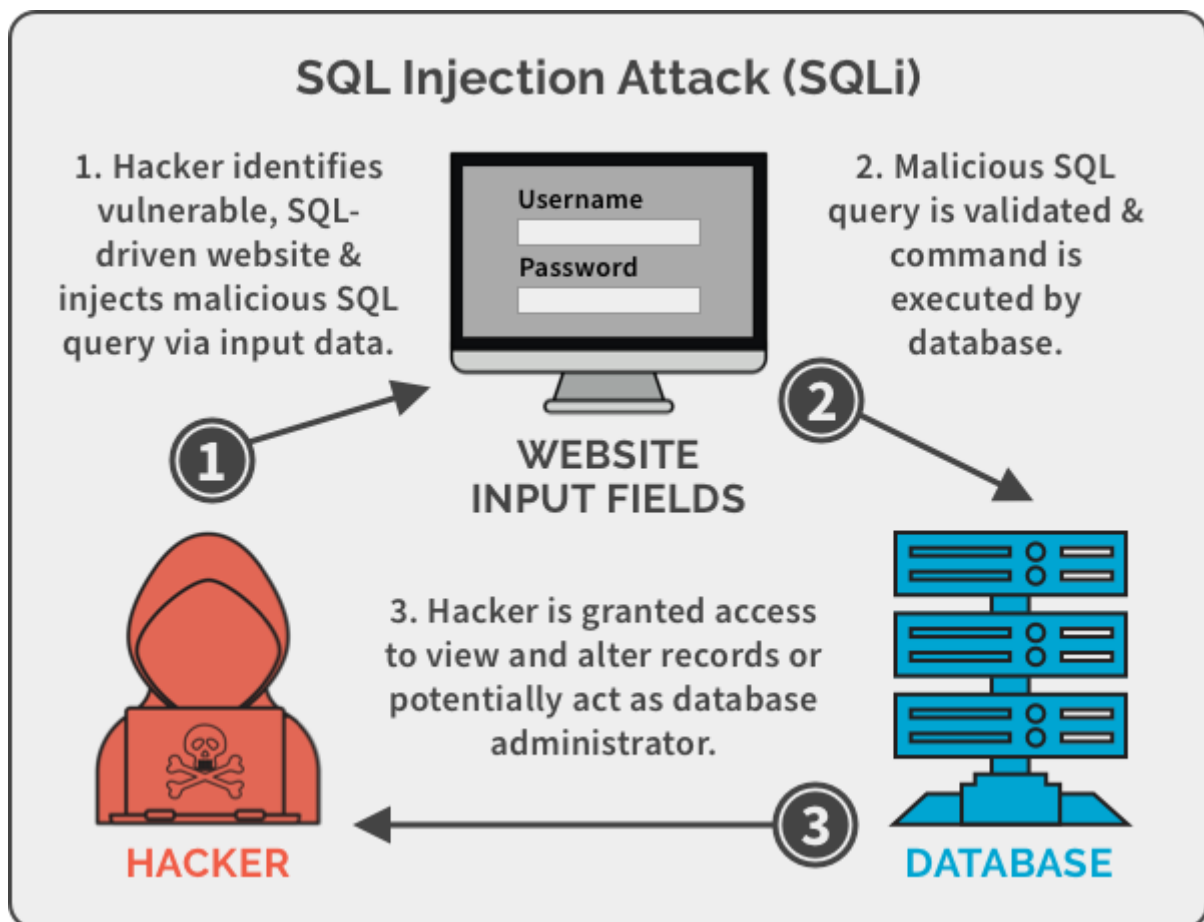
```
{
  "controls" :
  [
    {
      "controlId" : "TEST_OPEN",
      "applicationName" : ["test", "test_bh_control"],
      "events" : [
        {
          "eventId" : "sys_open",
          "action" : "TRACK",
          "evtargs" : ["foo.txt", "O_CREAT"],
          "fline" :
            [
              { "tName" : "test"
              }
            ]
        },
        {
          "eventId" : "sys_write",
          "action" : "FAIL",
          "evtargs" : ["SELECT *"],
          "fline" :
            [
              { "tName" : "test"
              }
            ]
        }
      ]
    }
  ]
}
```

- We now have an array of events which specify the actions to be performed on the various syscalls in the sequence.
- There is a new action, called TRACK, which essentially tracks the current syscall and expects a subsequent syscall for enforcement
- The evtargs list specifies the different arguments pertaining to the syscall which needs to be monitored.

We now look at how such types of controls are useful, in the context of an SQL injection.

SQL injection

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database. SQL injection is a form of web security flaw that allows an attacker to interfere with a web application's database queries. It helps an intruder to see data that they wouldn't usually be able to see. This may include data belonging to other users or any other information that the app has access to. In certain instances, an attacker may manipulate or erase this data, causing the application's content or behaviour to be permanently altered.



A simple example of an SQL injection is as follows:

Consider an application that lets users log in with a username and password. If a user submits the username 'shivam' and the password 'abcd', the application checks the credentials by performing the following SQL query:

```
SELECT * FROM users WHERE username = 'shivam' AND password = 'abcd'
```

The login is valid if the query returns the information of a user. Otherwise, it will be denied. An intruder can use the SQL comment sequence -- to eliminate the password check from the WHERE clause of the query, allowing them to log in as any user without a password.

For example, submitting the username admin'-- and a blank password results in the following query:

```
SELECT * FROM users WHERE username = 'admin'--' AND password = ''
```

This query returns the user whose username is administrator and successfully logs the attacker in as that user. The attacker could even write another SELECT query using boolean operators.

Such kinds of malicious queries could be fed into the system and lead to unauthorized access of information. To mitigate such attacks, we generate a control as shown above which specifies:

1. We monitor an open() syscall followed by a write() syscall.
2. The applications to be monitored are 'test_bh_control' and 'test'.
3. The file to be opened and modified is 'foo.txt' in 'O_CREAT' mode.
4. For this file, with given thread information, if the string 'SELECT *' is being written to file, fail this syscall.

In the state machine, we specify various flags that give us information about the current state of the file. If the control in question does not follow the state machine logic, it is considered invalid and then discarded. For e.g the syscalls may not be in the proper order. i.e write() or read() before open().

Use of the evtargs list which lists the various arguments of a syscall helps in designing controls in a more granular manner than before, so we can enforce our policies on a small subset of instances of a given syscall.

Whenever a file is opened, it is assigned a file descriptor integer, which is unique to it in a given process context. Making use of this fact, we understand that the (fd, pld) pair is unique for a given file, where

- fd is the file descriptor
- pld is the process id of the current process

So, we can use this pair in conjunction with the kretprobe as the key for our hash table for faster operation.

In summary, this expanded version of our control gives us a lot of information that helps us to model application processes more accurately. It also helps us to enforce our behaviour control policies more efficiently. The state machine logic may be expanded to include even more syscalls and modes of operation to make a more robust and well rounded mechanism for behaviour control. The challenge remains on how to characterise malicious software more accurately and deterministically, since various programs may make use of different pathways to influence execution. This would help us to generate better controls so as to make full use of our methodology.

References:

- <https://web.archive.org/web/20181009050551/http://cobweb.cs.uga.edu:80/~kyuhlee/publications/ndss13.pdf>
- <https://man7.org/linux/man-pages/man2/syscalls.2.html>
- <https://man7.org/linux/man-pages/man2/syscall.2.html>
- <https://www.kernel.org/doc/Documentation/kprobes.txt>
- https://uclibc.org/docs/psABI-x86_64.pdf
- <https://www.kernel.org/doc/html/latest/>
- <https://www.netsparker.com/blog/web-security/remote-code-evaluation-execution/>
- https://owasp.org/www-community/attacks/SQL_Injection