World Academy of Science, Engineering and Technology
International Journal of Bioengineering and Life Sciences
Vol:8, No:9, 2014

The Journey of a Malicious HTTP Request

M. Mansouri, P. Jaklitsch, E. Teiniker

***Abstract*—**SQL injection on web applications is a very popular kind of attack. There are mechanisms such as intrusion detection systems in order to detect this attack. These strategies often rely on techniques implemented at high layers of the application but do not consider the low level of system calls. The problem of only considering the high level perspective is that an attacker can circumvent the detection tools using certain techniques such as URL encoding. One technique currently used for detecting low-level attacks on privileged processes is the tracing of system calls. System calls act as a single gate to the Operating System (OS) kernel; they allow catching the critical data at an appropriate level of detail. Our basic assumption is that any type of application, be it a system service, utility program or Web application, "speaks" the language of system calls when having a conversation with the OS kernel. At this level we can see the actual attack while it is happening. We conduct an experiment in order to demonstrate the suitability of system call analysis for detecting SQL injection. We are able to detect the attack. Therefore we conclude that system calls are not only powerful in detecting low-level attacks but that they also enable us to detect high-level attacks such as SQL injection.

***Keywords*—**Linux system calls, Web attack detection, Interception.

I. Introduction

WEB applications are widely adopted as well as easily accessible. Therefore they are a popular target for attacks. For this reason initiatives devoted to providing Web application security such as OWASP[1] have become active. Among OWASP's contributions to Web application security there is an enumeration of the most dangerous attacks called OWASP Top Ten [9]. In this classification "Injection" attacks are at the top rank. Injection attacks include SQL injection where the attacker injects SQL commands in the data section of a query. Preventing such attacks is not always possible and even if there exist methods for effective prevention these methods might be disabled by developers due to a lack of security training or a lack of time in order to finish their task on schedule. This is the reason why "prevention mechanisms should be complemented by effective intrusion detection systems (IDSs)" [3].

In this paper we demonstrate how system call analysis can assist us in detecting an SQL injection attack. System call analysis as a means to detect attacks is primarily used for privileged applications such as Sendmail and lpr [2]. However, not only low-level applications invoke system calls to perform privileged tasks. For example a Java Web application is basically a Java program executed within a Java Virtual Machine (JVM) which translates the byte code into native code. Same as for system applications every privileged action performed by a Java program results in a system call being executed on the OS. Consequently we assume that there is no possibility for the data to bypass this interface between the native code of our application and the OS. As system calls operate at the kernel level they can provide us with information at a fine level of detail.

We start our demonstration using a simple Java Web application vulnerable to SQL injection. The data which is traveling through the different layers of our application from the highest level to the lowest level can be compared to a journey. Hence we structure our work from the perspective of a traveler.

In Section II we distinguish our work or journey from the previous ones. In Section III we first prepare our journey by considering our application from a high-level and from a low-level perspective or map. Subsequently we give an overview of the building blocks of our application or travel stations. The core part of this section is an experiment where we analyze the system calls invoked by our Web application that have been traced during an SQL injection attack. Due to the large amount of system calls we only select those relevant for our purpose. This can be compared to a travel diary where we only record the interesting experiences encountered during our journey. When we reach the destination of our journey we briefly discuss what we have discovered. In Section IV we conclude our paper or share our discoveries.

II. Related Work

The work we present in this paper is a combination of two perspectives for attack detection: A low-level perspective using system call tracing and a high-level perspective considering high-level attacks such as Web application attacks. In the following, we discuss how previous works in these two areas relate to our research work.

A. Low-Level Perspective

Bernaschi et al. [1] have demonstrated how the illegal execution of privileged operations can be prevented via minor amendments made to kernel code. Based on an in-depth system call analysis they identify a subset of system calls along with a subset of tasks that are helpful in preventing elevation of privilege attacks. They furthermore use this as input for developing an attack prevention prototype for the Linux OS implemented as a kernel module as well as a kernel patch. Once an attack is discovered the prototype denies further access requested by the particular process. The results of the analysis can assist in reducing the overhead inherent in

Malihe Mansouri and Paul Jaklitsch are Students at University of Applied Sciences FH JOANNEUM, Kapfenberg, Austria (e-mail: Malihe.Mansouri.ASE11@fh-joanneum.at, Paul.Jaklitsch.ASE10@fh-joanneum.at).

Egon Teiniker, FH Professor at University of Applied Sciences FH JOANNEUM, Kapfenberg, Austria (e-mail: Egon.Teiniker@fh-joanneum.at).

[1] Open Web Application Security Project.

attack monitoring as well as the effort for developing such a solution. It also helps reducing the efforts for developing more secure privileged applications.

Forrest et al. [2] have proposed a method for detecting anomalous behavior of privileged Unix processes. They adopt the idea of the immune system which distinguishes between self and others with 'self' defining normal behavior whereas 'others' refers to anomalous behavior or attacks. They define such as 'self' using short sequences of system calls making it efficient to monitor in terms of computing time. The authors demonstrate that the space of possible system call sequences remains fairly limited and that the sequences are closely related to the kind of process. Furthermore they have shown that it is very likely that the sequence is disturbed in case of an attack happening thus allowing for its detection. Their method was able to detect a number of known attacks on the UNIX programs Sendmail and lpr.

In their work Peisert et al. [5] go one level beyond tracing system calls by considering the usefulness of tracing function calls in order to assist in forensic analysis. They demonstrate that function calls provide a suitable level of abstraction for forensic analysts due to the fact that they provide an intuitive description when analyzing an attack. The authors note that anomalous sequences of function calls fulfill a basic requirement of forensics stating that besides the importance of being aware if an attack has occurred it is also important to know where it has occurred. Via conducting various experiments with exploits on privileged UNIX applications, they show that observing the deviations in the sequences of function calls help in detecting illegal process activities.

B. High-Level Perspective

Robertson et al. [3] have suggested a new approach for performing anomaly-based detection of web-based attacks with the goal to reduce the number of false positives generated by ordinary anomaly-based systems. Additionally they aim to provide the person responsible for responding to the attack with a description of the attack that has caused the anomaly. For this purpose they have developed a prototype of a web intrusion detection system. Their approach uses a technique for generalizing anomalies by turning suspicious HTTP requests into anomaly signatures. The event source for obtaining the request data is flexible and the prototype uses web server access log files. Once obtained the anomaly signatures serve as the grouping criterion for repeating or similar abnormal requests, facilitating the task of the administrator who has to respond to the alerts. Furthermore the approach uses a heuristics-based technique to determine the type of attack which caused the anomaly. Through this the attacks are prioritized and enriched with explanatory information. In order to evaluate their approach the authors have provided the system with real-world data stemming from access log files from web servers hosted at different universities.

Kruegel et al. [4] have introduced a novel approach for anomaly detection in HTTP requests. The event source providing the data for the analysis are web server access log files containing the HTTP requests together with the parameters. They implement analysis techniques which compare the access patterns of HTTP requests together with the contained parameters to profiles belonging to the requested server-side program or document that have been defined earlier. This allows the system to perform a focused analysis and to reduce the number of false positives. The system has been tested on real-world data as well.

C. Detecting High-Level Attacks

Considering the previous works we identify investigations on the usage of system calls for attack detection. In other words these methods use system calls as an event source. Others use network traffic [6] or web server log files to detect web-based attacks [3], [5].

To date we are not aware of any work dealing with the combination of system calls for attack detection and the goal of detecting web-based attacks. Robertson et al. [3] mention in a side note when describing the event collection component of their web intrusion detection system that events could also be collected from "a system call auditing facility embedded into a web server's host operating system" but they do not elaborate further on it. In our work we demonstrate the value of performing a system call analysis for the purpose of detecting Web application attacks. In order to perform a highly privileged task any user, or process, has to invoke system calls which are the only interface between user space and kernel space. Peisert et al. [5] encourage our idea by stating that "Capturing behaviors represented at the system call abstraction makes intuitive sense: Most malicious things an intruder will do use system calls."

III. The Journey of a Malicious HTTP Request

Before going on a journey we usually plan the route by looking at the desired destinations that we plan to visit and we prepare ourselves by gathering the required equipment. During our journey we want to share our discoveries with others.

A. Preparation of the Journey

In the following we discuss the application of our detection mechanism at different levels of the request life cycle. We do so by following an HTTP request containing input data with an SQL injection. Therefore we first want to introduce the different stations that we will visit during our journey at the application and system call level.

1. Stations of the Journey at Application Level

In the following we give a short overview of what happens to our HTTP request on its way to the OS level. We address the different stations using the numbers shown in Fig. 1. We have a simple Java Web application which receives the malicious query and passes it through different travel stations. After visiting the whole destinations it returns a response to the starting point of the journey which is the HTTP client.

On submission of the login form the client sends an HTTP request to the Apache Tomcat server. Then Tomcat forwards the request to the configured Filter Authentication Filter (1)

which performs authentication. In order to authenticate the user, the Filter delegates the login method to the business layer, in our concrete example to the class User Management (2). User Management forwards to the class User DAO (3) from the data layer to obtain the actual credentials from a MySQL database. Subsequently User Management compares the user provided credentials to the credentials from the database. If the authentication is successful then the request will be forwarded to the Login Servlet (4). If authentication is not successful then the Filter sends an HTML error page to the client.

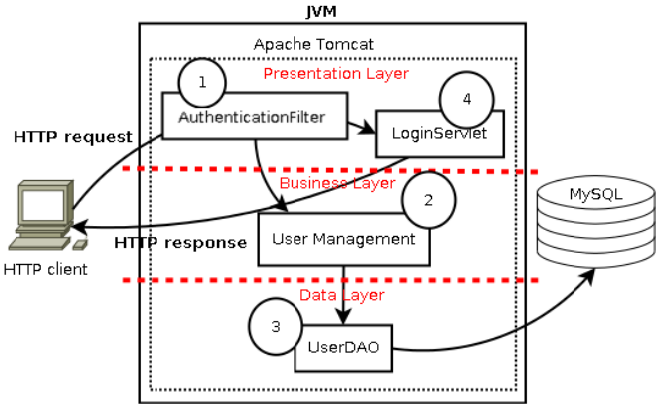

Fig. 1 Stations of HTTP request journey at application level view

2. Stations of the Journey at System Call Level

Fig. 2 shows the system calls for responding to an HTTP request. Tomcat receives the request data from the socket via the recv () system call (1) and then forwards the request to Authentication Filter. The latter attempts to authenticate the client and therefore calls the login method implemented by User Management. User Management obtains user information from the database via User DAO. User DAO accesses the MySQL database via JDBC. The JDBC driver first has to be authenticated at the MySQL server by sending username and hashed password using the send () system call (2). After authentication the JDBC driver uses the send () system call to transmit the malicious query to the database server (3). MySQL server returns the result of the query to the JDBC driver which obtains it using the recv () system call (5). If User DAO finds a user in the database with the corresponding username it returns a User object to User Management. Finally User Management compares the password of the user from database to the password provided in the request. If the authentication was successful Login Servlet sends the response to the client, otherwise Authentication Filter sends an error page. This will be done using the send () system call (5).

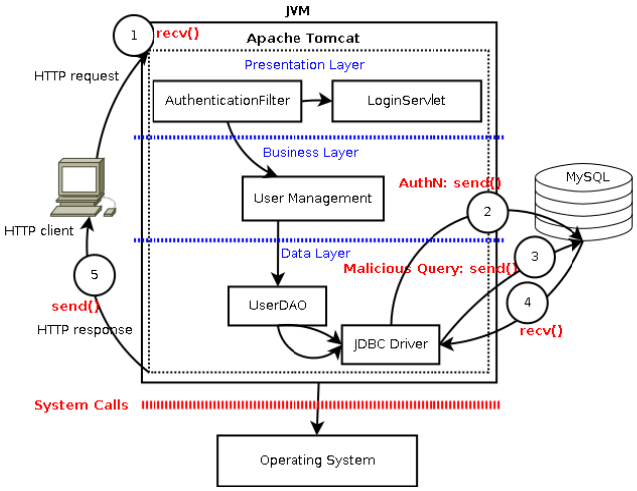

Fig. 2 Stations of HTTP request journey at system call level view

3. Participants of the Request Journey

Until now we assumed an ordinary user entering a valid password. It becomes even more interesting if the user injects a malicious query to perform an SQL injection attack. Therefore we want to detail the behavior of the application described above when it is under an SQL injection attack.

Our journey starts at the client side with the client sending an HTTP request from a simple HTML login form used as presentation as shown in Fig. 3.

```
<form method="get" action="/Servlet-Login/login.html" >
  <input type="text" name="username" maxlength="80" size="20">
  <input type="password" name="password" maxlength="40"
         size="20">
  <input type="submit" name="action" value="Login">
       …
</form>
```

Fig. 3 Index.html Page

This HTTP request is received by the Servlet[2] container. Tomcat in turn passes the request to the configured Filter Chain[3] which in our example consist of only one Filter used to perform authentication namely Authentication Filter. This preprocessing Filter allows to transparently authenticate the user and therefore acts as some kind of decorator of proxy which performs authentication functionality before generating the dynamic response. Only if authentication is successful the request will be forwarded to the Servlet which generates the response to the client. Circumventing this authentication mechanism is the target of SQL injection.

The Authentication Filter makes use of login functionality provided by the business layer. The class User Management belonging to the business layer is responsible for comparing the pair of user credentials as provided by the user to the credential stored in to the database. The class User Management uses an object of User DAO in order to obtain

[2] Java class creating dynamic web content that is capable of the HTTP protocol.

[3] A web resource can be filtered by a chain of 0 to n filters which are called in a predefined order. Among the applications of filters are authentication, logging and encryption [10].

the user credentials from the database. The implementation of the User DAO uses the Java JDBC[4] interface for connecting to a MySQL database. Due to the fact that the SQL statement is created from a fixed string concatenated with unsanitized user input attackers can choose their own password when providing a malicious SQL statement as input for the username in the login form.

B. Request Journey - Tracing SQL Injection

In the following section we analyze a trace of system calls which were performed by our application while being under an SQL injection attack. For this purpose we first need to trace the system calls invoked by our application. We consider the system calls resulting from our Web application running on Catalina[5] acting as the database client. We conduct our experiment on a Fedora Linux system using strace as a tracing tool which we consider to be both sufficiently accurate and simple to use for our purposes[6].

Considering we want to trace an instance of Apache Tomcat having a pid of 5124[7] and printing argument strings up to 1024 characters we use the following command (Fig. 4).

```
# strace –p 5124 –f -s 1024 –o traceServletApplication.txt
```

Fig. 4 Strace for Catalina process

1. Reducing the Set of Data

The resulting output file of our application contains a trace of more than 10000 lines of system calls invoked by the process where we attached strace. Same as in a journey when we record our experiences in a travel diary we have to distinguish between the interesting and the unnecessary information. In order to reduce the number of lines that we have to analyze we look for a classification which can give us the exact group of system calls which are relevant to our client/server application. Bernaschi et al. [1] provide a system call classification based on functionality. In terms of this classification our application invokes system calls belonging to the groups of communication, more precisely network communication, and file systems.

2. Travel Diary - System Call Analysis

As starting point we analyze the request to the static page index.html. Subsequently Tomcat has to process this request and therefore needs to open the requested file, read the content and send it to the client. In the following we identify the individual system calls for receiving the client request at the server side as depicted in Fig. 5.

Each line in the trace file starts with the process id executing the system call, in this case “7480” which is the process id of the Java virtual machine. Therefore we can reduce further the set of system calls by only considering system calls related to this process. The recv() system call is used to receive messages from a connected socket. The argument “41” refers to the file descriptor of the socket, the second argument corresponds to the buffer contents i.e. the HTTP request, the third argument “8192” describes the buffer length in bytes and the last argument “0” is a flag for defining the desired behavior. The return value “317” indicates the number of bytes received from the socket.

```
…
7480 <... gettimeofday resumed> {1386544364, 110928}, NULL) = 0
7480 poll([{fd=41, events=POLLIN|POLLERR}], 1, 20000) = 1 ([{fd=41, revents=POLLIN}])
7480 recv (41, "GET /Servlet-Login/index.html HTTP/1.1\...\n", 8192, 0)=317
7480 gettimeofday({1386544364, 111946}, NULL) = 0
7480 stat64("/home/student/install/apache-tomcat7.0.26/wtpwebapps/Servlet-Login/index.html", {st_mode=S_IFREG|0664, st_size=1247, ...}) = 0
…
```

Fig. 5 System call receiving client request

The next relevant system call stat64() lists information about the file index.html (Fig. 6).

```
…
7480 recv(41, "GET /Servlet-Login/index.html \...\n", 8192, 0) = 317
7480 gettimeofday({1386544364, 111567}, NULL) = 0
7480 gettimeofday({1386544364, 111946}, NULL) = 0
7480 stat64("/home/student/install/apache-tomcat-7.0.26/wtpwebapps/Servlet-Login/index.html", {st_mode=S_IFREG|0664, st_size=1247, ...}) = 0
7480 access("/home/student/install/apache-tomcat-.0.26/wtpwebapps/Servlet-Login/index.html", R_OK) = 0
7480 lstat64("/home", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
7480 lstat64("/home/…", {st_mode=S_IFDIR|0700, st_size=4096, ...}) = 0…
```

Fig. 6 System call list file information

The first argument is the file path, and the second argument constitutes a buffer to be filled with the information such as the protection of the regular file *st_mode* and *st_size* showing the file’s total size.

Having the file information Tomcat invokes the “access” system call in order to check if all requested permissions are granted.

The second argument to access() requests a check if the file exists and read permissions are granted to it which is the case according to the return value “0” (Fig. 7).

```
…
7480 stat64("/home/student/install/apache-tomcat-.0.26/wtpwebapps/Servlet-Login/index.html", {st_mode=S_IFREG|0664, st_size=1247, ...}) = 0
7480 access ("/home/student/install/…/wtpwebapps/Servlet-Login/index.html", R_OK) = 0
7480 lstat64("/home", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
7480 lstat64("/home/…", {st_mode=S_IFDIR|0700, st_size=4096, ...}) = 0
…
```

Fig. 7 System call file permission check

The open () system call returns a file handle to index.html that will be used by the subsequent read() system call (Fig. 8).

[4] Java Database Connectivity.
[5] Tomcat's servlet container implementation is called Catalina.
[6] For further information about strace refer to the Appendix.
[7] We have to determine the pid related to the script catalina.sh, not startup.sh.

```
…
7480  stat64("/home/student/install/apache-tomcat-.0.26/wtpwebapps/Servlet-Login/index.html", {st_mode=S_IFREG|0664, st_size=1247, ...}) = 0
7480 open ("/home/student/…/wtpwebapps/Servlet-Login/index.html", O_RDONLY|O_LARGEFILE) = 42
7480  fstat64(42, {st_mode=S_IFREG|0664, st_size=1247, ...}) = 0
7480  fcntl64(42, F_GETFD) = 0
…
```

Fig. 8 System call open() file handle

The following read () reads the entire content of index.html from file descriptor "42" (Fig. 9).

```
…
7480  fcntl64(42, F_GETFD) = 0
7480  fcntl64(42, F_SETFD, FD_CLOEXEC) = 0
7480 read (42, "<html>\n\t<head>\n\t\t<title>Servlet Filter Example</title>\n\t</head>\n\t<body>...\n\t\t<h2>Login: </h2>\t\t\n\t\t<form method=\"get\" action=\"/Servlet-Login/login.html\" >….", 1247) = 1247
7480  close(42) = 0
7480  gettimeofday({1386544364, 118267}, NULL) = 0
…
```

Fig. 9 System call read()

The server now creates an HTTP response using the file recently read together with header information and sends it to the client via the socket identified by file descriptor "41" (Fig. 10).

```
…
7480  gettimeofday({1386544364, 119761}, NULL) = 0
7480  gettimeofday({1386544364, 119903}, NULL) = 0
7480 send (41, "HTTP/1.1 200 OK\r\nServer: Apache-Coyote/1.1\r\nAccept-Ranges: bytes\r\nETag: W/\"1247-1386504489000\"\r\nLast-Modified: Sun, 08 Dec 2013 12:08:09 GMT\r\nContent-Type: text/html\r\nContent-Length: 1247\r\nDate: Sun, 08 Dec 2013 23:12:44 GMT\r\n\r\n<html>\n\t<head>\n\t\t<title>Servlet Filter Example</title>…
", 1475, 0 <unfinished ...>
5131  <... gettimeofday resumed> {1386544364, 110930}, NULL) = 0
7480  <... send resumed> ) = 1475
5131  clock_gettime(CLOCK_REALTIME,  <unfinished ...>
…
```

Fig. 10 System call send()

As shown in Fig. 11 the attacker enters the malicious SQL string in order to obtain access to the application and submits the request to the server by pressing the "Login" button.

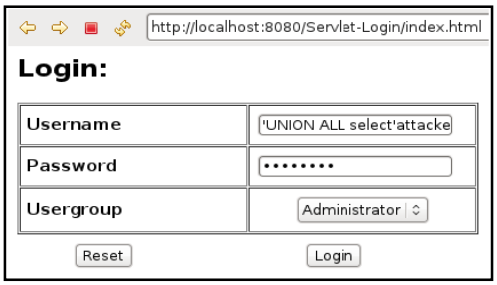

Fig. 11 SQL injection login form

Until now we just had the static page but shortly the attacker will enter the malicious data in the field of Username and Password using the values depicted in Fig. 12.

Username: 'UNION ALL select 'attacker','Pa$$w0rd'from dual where "='
Password: Pa$$w0rd

Fig. 12 Malicious Query

Subsequently Tomcat receives the username and password from the client in URL encoded form (Fig. 13)

```
…
7483  poll([{fd=41, events=POLLIN|POLLERR}], 1, 20000 <unfinished ...>
5131  futex(0xb772a444, FUTEX_WAIT_PRIVATE, 5, {0, 999835953} <unfinished ...>
7483  <... poll resumed> ) = 1 ([{fd=41, revents=POLLIN}])
7483 recv (41, "GET /Servlet-Login/login.html?username=%27UNION+ALL+select%27attacker%27%2C%27Pa%24%24w0rd%27from+dual+where+%27%27%3D%27&password=Pa%24%24w0rd&usergroup=Guest&action=Login HTTP/1.1\r\nHost: localhost:8080\r\nUser-Agent: Mozilla/5.0 (X11; Linux i686) AppleWebKit/535.4+ (KHTML, like Gecko) Version/5.0 Safari/535.4+\r\nAccept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\nReferer: http://localhost:8080/Servlet-Login/index.html\r\nAccept-Encoding: gzip\r\nConnection: Keep-Alive\r\n\r\n", 8192, 0) = 492
7483  gettimeofday({1386544409, 846053}, NULL) = 0
7483  write(1, "SQL> SELECT username, password FROM User WHERE username ="UNION ALL select'attacker','Pa$$w0rd'from dual where "='", 117) = 117
7483  write(1, "\n", 1)  = 1
   …
```

Fig. 13 Receive malicious query

Tomcat after receiving these parameters and values has to do a dynamic processing. The login.html requires that the user will be authenticated. Therefore the credential data provided by the user must be compared to the credentials stored into the database and in order to do that our User DAO uses a JDBC driver to communicate with the MySQL database.

```
…
7483  gettimeofday({1386544409, 852614}, NULL) = 0
7483 socket (PF_INET6, SOCK_STREAM, IPPROTO_IP) = 42
7483 setsockopt (42, SOL_IPV6, IPV6_V6ONLY, [0], 4) = 0
7483  connect(42, {sa_family=AF_INET6, sin6_port=htons(3306), inet_pton(AF_INET6, "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, 28) = 0
7483  getsockname(42, {sa_family=AF_INET6, sin6_port=htons(48367), inet_pton(AF_INET6, "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=0, sin6_scope_id=0}, [28]) = 0
7483 setsockopt (42, SOL_TCP, TCP_NODELAY, [1], 4) = 0
7483 setsockopt (42, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0
7483  gettimeofday({1386544409, 856534}, NULL) = 0
…
```

Fig. 14 System call socket() connection

As shown in Fig. 14 subsequently the connector establishes a connection with the database server on port "3306"which in our experiment resides at the same machine (127.0.0.1).

Fig. 15 depicts how our application receives a response from the MySQL server. In this response the MySQL server tells the client to use a password mechanism called

"*mysql_native_password*, which implements authentication against the mysql.user table using the native password hashing method [8].

```
…
7483 setsockopt(42, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0
7483 gettimeofday({1386544409, 856534}, NULL) = 0
7483 ioctl(42, FIONREAD, [78]) = 0
7483 recv(42, "J\0\0\0\n5.5.20\0\230\0\0\0R-(~ghUN\0\377\367\10\2\0\17\200\25\\0\0\0jHk`#*Ipv%O@\0mysql_native_password\0", 78, 0) = 78
7483 gettimeofday({1386544409, 856873}, NULL) = 0
…
```

Fig. 15 System call password negotiation

In the send() system call the JDBC driver connects to the database "testdb" using the user "student" and the hashed password which can be seen in Fig. 16.

```
...
7483 recv(42, "J\0\0\0\n5.5.20\...@\0mysql_native_password\0", 78, 0) = 78
7483 send (42, "D\0\0\1\217\242\2\0\377\377\377\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0student\0\24\252c\312v\360\253\302\362\\333\217\253testdb\0", 72, 0) = 72
5137 futex(0xb7740e28, FUTEX_WAKE_PRIVATE, 1) = 0
5137 gettimeofday({1386544409, 873247}, NULL) = 0
...
```

Fig. 16 MySQL client authentication

After multiple messages exchanged between the JDBC client and MySQL database server the client sends the malicious query to the server as shown in Fig. 17.

```
…
7483 gettimeofday({1386544409, 943173}, NULL) = 0
7483 ioctl(42, FIONREAD, [0]) = 0
7483 send (42, "q\0\0\0\3SELECT username, password FROM User WHERE username ="UNION ALL select'attacker','Pa$$w0rd'from dual where "="", 117, 0) = 117
7483 gettimeofday({1386544409, 961161}, NULL) = 0
7483 ioctl(42, FIONREAD, [0]) = 0
7483 recv(42, "\1\0\0\1", 4, 0) = 4
…
```

Fig. 17 Sending of malicious query

The second argument in the send() system call is the actual query at runtime that will finally be received and executed by MySQL. In this concrete example we can see the attack happening.

```
…
7483 <... ioctl resumed> , [125]) = 0
5137 clock_gettime(CLOCK_REALTIME, {1386544409, 977925904}) = 0
7483 recv (42, <unfinished ...>
5137 futex(0xb7740e44, FUTEX_WAIT_PRIVATE, 1, {0, 49902096}
<unfinished ...>
7483 <... recv resumed>"\2&\0\0\2\3def\0\0\0\10username\10username\f\10\0@\0\0\0\375\1\\0\5\0\0\4\376\0\0\"\0\22\0\0\5\10attacker\10Pa$$w0rd\5\0\0\6\376\0\0\"\0", 125, 0) = 125
7483 gettimeofday({1386544409, 978216}, NULL) = 0
7483 gettimeofday({1386544409, 978296}, NULL) = 0
…
```

Fig. 18 Result set from MySQL server

Finally the class Login Servlet generates an HTTP response and sends it to the attacker (Fig. 19).

```
…
5137 clock_gettime(CLOCK_REALTIME, {1386544410, 30823329}) = 0
5137 futex(0xb7740e44, FUTEX_WAIT_PRIVATE, 1, {0, 49942671}
<unfinished ...>
7483 gettimeofday({1386544410, 39904}, NULL) = 0
7483 send (41, "HTTP/1.1 200 OK\r\nServer: Apache-Coyote/1.1\r\nContent-Type: text/html;charset=ISO-8859-1\r\nContent-Length: 299\r\nDate: Sun, 08 Dec 2013 23:13:30 GMT\r\n\r\n<html> <head> <title>Login</title> </head> <body> <h1>Request Parameters:</h1> <pre> username = \"'UNION ALL select'attacker','Pa$$w0rd'from dual where "="\"<br/> password = \"Pa$$w0rd\"<br/> usergroup = \"Guest\"<br/> action = \"Login\"<br/> </pre> </body></html>\n", 447, 0) = 447
7483 gettimeofday({1386544410, 40707}, NULL) = 0
7483 gettimeofday({1386544410, 40802}, NULL) = 0
…
```

Fig. 19 HTTP response for login.html

Fig. 20 depicts that the attacker successfully gained access to the application.

Fig. 20 Successful attack

C. Findings of the Journey

So far we have considered the system calls invoked by our vulnerable application during an SQL injection attack. We can reduce the immense amount of traced system calls and focus on the interesting ones by prioritizing them according to their functionality [1] as well as the type of the application, such as a Web application that we used in our experiment. In our example we have been able to detect the SQL injection attack on the Web application by looking for the malicious query as an argument of the send() system call which is in the class of communication system calls. This is the point where the Web application, or more precisely the JDBC driver, sends the query to the database.

```
SELECT username, password FROM User WHERE username ="UNION ALL select'attacker','Pa$$w0rd'from dual where "="
```

Fig. 21 Malicious query

We can check for different attack patterns in the system call parameters *in a programming language independent way* and can concentrate on the peculiarities of the attack itself. (E.g. we do not need to know the mechanism used by the concrete programming language in order to access the DB). Based on the specifics of the attack we can define an attack signature. In our example the injected query string uses single quotes to satisfy the SQL syntax of the query which is defined in the data layer code (depicted in black in Fig. 21). One suspicious part of the query is that the user name part of the where clause is filtering for an empty string. This makes sure that the first

part of the statement has no result. The UNION ALL part of the SQL statement provides a WHERE condition which is always "true" and therefore the data is included into the result set. These are examples of suspicious patterns indicating that an SQL injection takes place.

Additionally we are able to check data that is exchanged between the components belonging to the infrastructure of a single application, including the server application, a web service or a database server. In our experiment the Web application communicates with the DB server using the send() and recv() system calls.

Furthermore the approach of detecting attacks via system calls applies not only to one specific attack like SQL injection but to all kinds of Web application attacks including cross-site scripting and path manipulation. The only pre-requisite is that we know about the attack pattern that we are looking for.

IV. Conclusion

Previous works [1], [2] focused on using system calls for detecting low level attacks which are targeting high privileged processes such as Sendmail. We could demonstrate a new way of using system calls for attack detection which enables us to detect SQL injection attacks which are taking place at a high level. System call tracing is applicable for detecting attacks independent from the level where they take place.

Moreover in order to detect the attack we do not need to have access to the application's source code.

Due to the fact that at runtime any application is represented as native code invoking the system calls we are independent from the programming language.

Furthermore we have no dependency on the application's internal architecture and whether it uses a layered architecture or not. Other approaches to attack detection might rely on a certain architecture e.g. in order to instrument existing code with log statements.

Last but not least is that we are able to see the actual data at runtime after it has passed any decoding and decryption which allows us to see the attack data in detail and which makes certain evasion techniques used by the attacker useless.

Appendix

Tracing system calls via strace can be described as "*a technique that presents details of the execution of program*". Following the path of a programs execution enables us to more accurately understand how a program executes and thereby interacts with its environment. Additionally following the path of execution enables us to detect the locations where the program does not behave as expected [7]. To this end for our work we need a tool which accurately shows the system calls which execute from the application that we want to investigate.

Strace is a tool allowing to trace system calls performed by a process and can either be attached to a running process (using the option –p) or be started with a new process.

When performing its task it records the system calls made by a process as well as the signals it receives.

For each system call it records the name, its arguments and the return value.

Strace does not require that the source code is available as it does not require any recompilation.

Child processes that have been created by a forked system call can be traced using the –f option and the output can be redirected to a file using the –o option if desired.

We should keep in mind that strace always has to be run with root privileges in order to also trace privileged system calls.

References

[1] M. Bernaschi, "Remus: a security-enhanced operating system," ACM Trans. on Information and System Security (TISSEC), 2002, pp.36-61.

[2] S. Forrest, S. A. Hofmeyr, "A Sense of Self for Unix Processes," *in Proc. IEEE Symposium on Security and Privacy*, Washington, 1996, pp. 120.

[3] W. Robertson, G. Vigna, "Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks, "in *Proc. of the 13th Symposium on Network and Distributed System Security* California, 2006.

[4] C. Kruegel, G. Vigna, "A multi-model approach to the detection of web-based attacks," *Elsevier Computer Networks: The International Journal of Computer and Telecommunications Networking - Web security*, New York, 2005,pp. 717 - 738.

[5] S. Peisert, M. Bishop, S. Karin, and K. Marzullo, "Analysis of Computer Intrusions Using Sequences of Function Calls," in *IEEE Trans. on Dependable and Secure Computing*, 2007, 137-150.

[6] Gustavo Miguel Barroso Assis do Nascimento, "Anomaly detection of web-based attacks," Master Thesis. Lisboa, Portugal, Universidade de Lisboa, 2010.

[7] M. T. Jones, IBM, "Kernel command using Linux system calls," from http://www.ibm.com/developerworks/linux/library/l-system-calls, 2010, Retrieved 12 11, 2013.

[8] Oracle, "The Native Authentication Plug-in," from http://dev.mysql.com/doc/refman/5.5/en/native-authentication-plugin.html, 2013, Retrieved 12 11, 2013.

[9] OWASP, "2013 Top 10 List", from https://www.owasp.org/index.php/Top_10_2013-Top_10, Retrieved 9 14, 2014

[10] Oracle, "Chapter 4 Java Servlet Technology: Filtering Requests and Responses", http://docs.oracle.com/cd/E19159-01/819-3669/bnafd/index.html