Security Vulnerability Notice

SE-2014-01-ORACLE

[Security vulnerabilities in Oracle Database Java VM, Issues 1-20]

DISCLAIMER

Security Explorations discovered multiple security vulnerabilities in the implementation of a Java VM embedded by Oracle Database software. Below, we provide technical details of our findings in a form of two sections. The first section outlines the weaknesses uncovered. The second one describes privilege elevation techniques for gaining DBA role privileges in a target database upon a successful compromise of its incorporated Java VM.

1. VULNERABILITIES

Almost all of discovered security vulnerabilities are the result of insecure implementation of Java Reflection API. Their successful exploitation can easily lead to the full compromise of a Java security sandbox of a target database server instance. This can further be exploited to execute arbitrary OS commands on a database server or to gain administrator privileges in a target Oracle Database.

A table below presents a summary of discovered Java security issues:

ISSUE #	TECHNICAL DETAILS	
1	origin	`oracle.jpub.reflect.RObject`
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation
	type	complete security bypass vulnerability
2	origin	`oracle.jpub.reflect.RMethod`
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation
	type	partial security bypass vulnerability
3	origin	`oracle.jpub.reflect.RClass`
	cause	insecure use of `getDeclaredMethod()` method of `java.lang.Class` class
	impact	access to declared methods of arbitrary classes
	type	partial security bypass vulnerability
4	origin	`oracle.sqlj.runtime.OraDynamicClosure`
	cause	insecure use of `invoke` method of `java.lang.reflect.Method` class
	impact	arbitrary method invocation
	type	complete security bypass vulnerability
5	origin	`oracle.jpub.reflect.RClass`
	cause	insecure use of `forName()` method of `java.lang.Class` class
	impact	access to arbitrary classes
	type	partial security bypass vulnerability
6	origin	`oracle.jpub.reflect.RClass`
	cause	insecure use of `getDeclaredConstructor()` method of `java.lang.Class` class
	impact	access to declared constructors of arbitrary classes
	type	partial security bypass vulnerability
7	origin	`oracle.jpub.reflect.RClass`
	cause	insecure use of `getMethod()` method of `java.lang.Class` class
	impact	access to methods of arbitrary classes
	type	partial security bypass vulnerability
8	origin	`oracle.jpub.reflect.RClass`
	cause	insecure use of `forName()` method of `java.lang.Class` class
	impact	access to arbitrary classes
	type	partial security bypass vulnerability
9	origin	`oracle.jpub.reflect.RClass`
	cause	insecure use of `getDeclaredConstructor()` method of

		`java.lang.Class` class
	impact	access to declared constructors of arbitrary classes
	type	partial security bypass vulnerability
10	origin	`oracle.jpub.reflect.RClass`
	cause	insecure use of `getMethod()` method of `java.lang.Class` class
	impact	access to methods of arbitrary classes
	type	partial security bypass vulnerability
11	origin	`oracle.aurora.util.ClassDescription`
	cause	insecure use of `getDeclaredFields()` method of `java.lang.Class` class
	impact	access to declared fields of arbitrary classes
	type	partial security bypass vulnerability
12	origin	`oracle.aurora.util.JRIExtensions`
	cause	insecure access bits of a native method
	impact	arbitrary field access
	type	partial security bypass vulnerability
13	origin	`oracle.sqlj.runtime.OraDynamicClosure`
	cause	insecure use of `getDeclaredMethod()` method of `java.lang.Class` class
	impact	access to declared methods of arbitrary classes
	type	partial security bypass vulnerability
14	origin	`oracle.jpub.reflect.Server`
	cause	insecure use of `forName()` method of `java.lang.Class` class
	impact	access to arbitrary classes
	type	partial security bypass vulnerability
15	origin	`oracle.jpub.reflect.Server`
	cause	insecure use of `getConstructor()` method of `java.lang.Class` class
	impact	access to constructors of arbitrary classes
	type	partial security bypass vulnerability
16	origin	`oracle.sqlj.runtime.OraDynamicClosure`
	cause	insecure use of `getDeclaredMethod()` method of `java.lang.Class` class
	impact	access to declared methods of arbitrary classes
	type	partial security bypass vulnerability
17	origin	`oracle.sqlj.runtime.OraDynamicClosure`
	cause	insecure use of `forName()` method of `java.lang.Class` class
	impact	access to arbitrary classes
	type	partial security bypass vulnerability
18	origin	`sqlj.runtime.profile.ref.IterConvertProfile.IterConvertStatement`
	cause	insecure use of `getDeclaredConstructor()` method of `java.lang.Class` class
	impact	access to declared constructors of arbitrary classes
	type	partial security bypass vulnerability
19	origin	`oracle.sqlj.runtime.OraDynamicClosure`
	cause	insecure use of `getMethod()` method of `java.lang.Class` class
	impact	access to methods of arbitrary classes
	type	partial security bypass vulnerability

Reported Java security sandbox bypass issues illustrate known and widely discussed security risks related to the use of Java SE Reflection API. They are caused by exactly the same violations of Oracle's own Secure Coding Guidelines [1] as we reported for Java SE [2] and Oracle Java Cloud Service [3].

Arbitrary Java class loading and execution (Issue 20)
According to [4], a user needs CREATE PROCEDURE privilege to define Java classes (source and binary) and resources in a target Oracle Database environment. The same

documentation specifies that LOADJAVA tool additionally requires CREATE TABLE privilege to load arbitrary classes into a database.

The above is true from a database point of view, but not necessarily from a Java VM execution engine perspective.

There exists a way for an unprivileged user (with a CREATE SESSION privilege only) to both "load" and execute arbitrary user provided classes in Oracle Database server. This can be accomplished with the use of the following features of Oracle Database Java VM:

- `dbms_java.start_loading_jar` and `dbms_java.finish_loading_jar` expose a functionality to load arbitrary JAR files (containing Java classes) into the Oracle Database. The content of loaded JAR objects is visible along the system JAR files in a system `JAVAJAR$` table,
- Oracle Database Java VM implements internal `jserver` URL protocol that allows to reference JAR objects and resources defined in user schemas. The content of JAR files loaded into the database can be accessed through `jserver` URL protocol (`jserver:/CP/JAR/SCHEMA/`*`schemaname`*`/`*`jarname`*)
- A classpath term beginning with the literal substring "`JSERVER_CP`" is converted to an URL by replacing JSERVER_CP with `jserver:/CP` [4],
- `dbms_java.runjava` allows for execution of arbitrary Java classes. The call can be provided with an argument denoting a classpath.

We verified that a user with a bare minimum CREATE SESSION database privilege can successfully "load" and execute arbitrary user provided classes in Oracle Database server.

The loaded classes are not fully defined in a database environment (i.e. invisible to ALTER JAVA or DROP JAVA constructs). This is however not an obstacle at all, as the possibility to load and execute malicious Java code is completely sufficient to successfully compromise the security of Oracle Database as explained in the next section of this paper.

Attached to this report, there are 8 Proof of Concept codes illustrating discovered weaknesses. Their successful exploitation allows for a complete Java security sandbox bypass in a target Oracle Database environment.

2. PRIVILEGE ELEVATION TECHNIQUES

Proof of Concept Codes accompanying the reported vulnerabilities implement 3 different exploitation techniques that allow to gain DBA role privileges in a target database upon a successful compromise of its embedded Java VM. These exploitation techniques are briefly described below.

Exploitation through `sqlplus` command execution

The most naive method for gaining database administrator privileges can be accomplished by the means of arbitrary `sqlplus` command execution. Escape of a Java security sandbox allows for execution of arbitrary binaries on a database server through `Runtime.exec()` method.

Spawned `sqlplus` process can be provided with arbitrary PL/SQL command input. The commands are executed with database administrator privileges if `sqlplus` process is started in DBA administrator mode:

```
sqlplus / as sysdba
```

The described exploitation technique might not be applicable if `dbms_java.set_runtime_exec_credentials` procedure is used to control OS user identities of spawned commands.

Exploitation through a privileged definer frame
Oracle database employs the idea of a DEFINER and CURRENT_USER authorization types for PL/SQL code. The same applies to Java methods. Methods of a class marked as AUTHID DEFINER execute with the privileges of a class owner. Methods of a class marked as AUTHID CURRENT_USER execute with the privileges of a caller. Internal identity stack structure helps evaluate the effective ID of the code for security verification purposes.

The security model implemented by Oracle Database lacks the advantages of the scoped privilege model with stack inspection [5] introduced into JDK 1.2 and Netscape 4.0 more than 15 years ago. In this model, a given privilege must be explicitly granted to the code requesting restricted operation, it must be explicitly enabled before a potentially harmful operation. Finally, it is valid only for the stack frame of the code that enabled it. By using privileges along with a stack inspection mechanism, the threat of escaping the Java applet sandbox through exploitation of some potentially vulnerable system class is drastically minimized. This is due to the fact that in Java all method frames from within a trusted call sequence (i.e. privileged method block) are inspected during privilege checking operation. In Oracle Database case, usually only the top of the identity stack matters. This means that an arbitrary call to attacker provided code (PL/SQL or Java marked as AUTHID CURRENT_USER) made from within the trusted call sequence will not affect the effective privileges seen by the database security engine. The reason for it is because only AUTHID DEFINER classes influence the content of the identity stack. This is illustrated on Fig. 1.

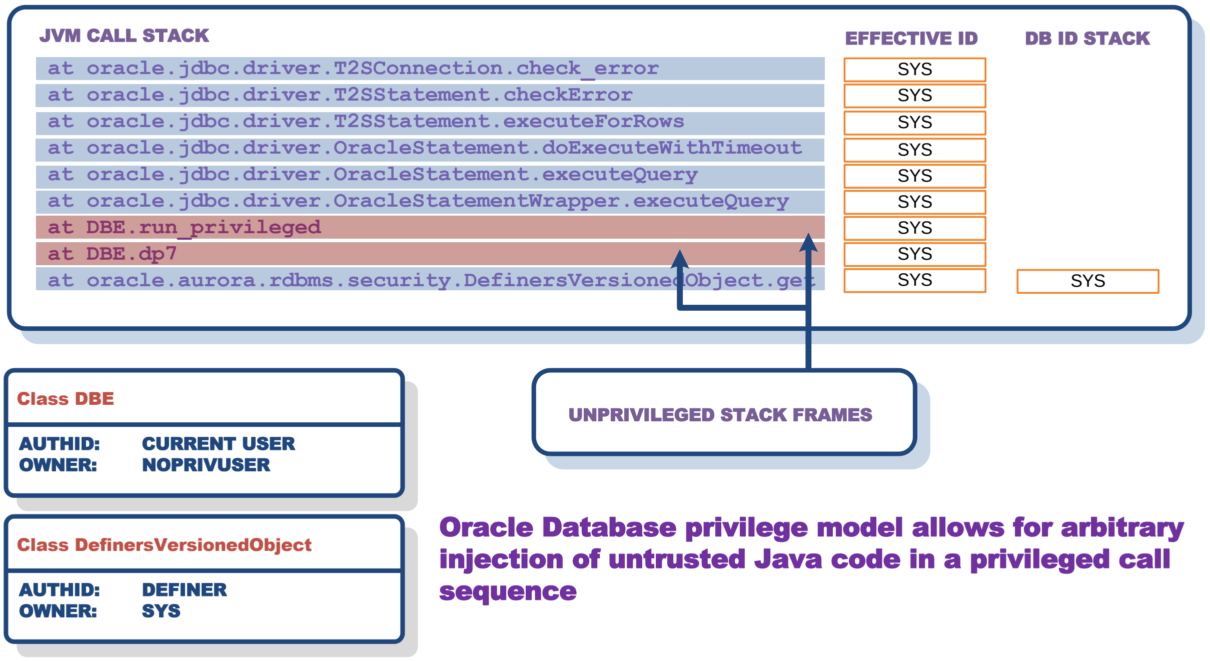
JVM CALL STACK
EFFECTIVE ID
DB ID STACK
at oracle.jdbc.driver.T2SConnection.check_error
at oracle.jdbc.driver.T2SStatement.checkError
at oracle.jdbc.driver.T2SStatement.executeForRows
at oracle.jdbc.driver.OracleStatement.doExecuteWithTimeout
at oracle.jdbc.driver.OracleStatement.executeQuery
at oracle.jdbc.driver.OracleStatementWrapper.executeQuery
at DBE.run_privileged
at DBE.dp7
at oracle.aurora.rdbms.security.DefinersVersionedObject.get
SYS
UNPRIVILEGED STACK FRAMES
Class DBE
AUTHID: CURRENT USER
OWNER: NOPRIVUSER
Class DefinersVersionedObject
AUTHID: DEFINER
OWNER: SYS
Oracle Database privilege model allows for arbitrary injection of untrusted Java code in a privileged call sequence

Fig. 1 Illustration of Oracle Database privilege model as seen by the embedded Java VM.

This deficiency of Oracle Database security model has been the subject of abuse for many years [6][7]. The majority of the techniques exploited vulnerabilities in a privileged PL/SQL code to inject arbitrary PL/SQL code of attacker's choice.

Below, we present 2 techniques that exploit the abovementioned weakness of Oracle Database security model through the embedded Java VM. Both of them can be successfully used to elevate privileges in Oracle Database environment upon a complete Java security sandbox bypass of the incorporated Java VM. Both techniques abuse the implementation of AUTHID DEFINER construct for database procedures and functions defined in a Java language.

Please note, that for the purpose of further discussion, the terms *AUTHID DEFINER class / definer class* or simply *definer* are used interchangeably.

Definer spoofing

Java VM runtime of Oracle Database maintains a dedicated stack of identities for a given execution context. This is the `eocontext_idstk` field of `struct eocontext`. Upon invocation of a given Java method represented internally by `struct jom_method`, its `jom_method_access` field is inspected to see whether a value corresponding to the method's owner should be pushed onto the identity stack:

```
           test    r12d, 10000h        ; is JOM_ACC_DEFINERS set ?
           jz      short loc_5BEB7B4
           mov     rcx, r15            ; eocontext *ctx
           mov     rdx, r13            ; jom_active_clint *id
           mov     r8, r14             ; eoforptr *frame
           call    _joet_push_id__
loc_5BEB7B4:
```

If `JOM_ACC_DEFINERS` flag is set in method access flags, the id of the method's owner gets pushed onto the identity stack.

The interesting things happen in `joet_push_id` code:

```
           mov     rbx, r8          ; frame
           mov     r15, rdx         ; jom_active_clint
           mov     rsi, rcx         ; ctx
           mov     rcx, rsi
           call    _joet_id_stack_top__
           mov     r10, rax         ; eocontext_idstk
           mov     r14, r15
           and     r14, 0FFFFFFFFFFFFFFF8h
           mov     rbp, [r14+28h]   ; jom_active_clint_pair
           test    r10, r10         ; eocontext_idstk
           jz      short loc_5BF9D1F
           test    rbp, rbp
           jz      short loc_5BF9D1F
           mov     rdx, r10         ; eocontext_idstk
           and     rdx, 0FFFFFFFFFFFFFFF8h
           cmp     rbp, [rdx+18h]   ; are eoidstk_pairs equal ?
           jz      loc_5BF9F03      ; jump to exit if equal
loc_5BF9D1F:
```

In the above code, `jom_active_clint_pair` field of `jom_active_clint` structure is compared to the top element of the identity stack. If they are equal (in terms of pointer's equivalence), no value is pushed onto the identity stack. This condition corresponds to the invocation of the method from the class of which owner has been already pushed onto the identity stack (effective ID does not change across calls).

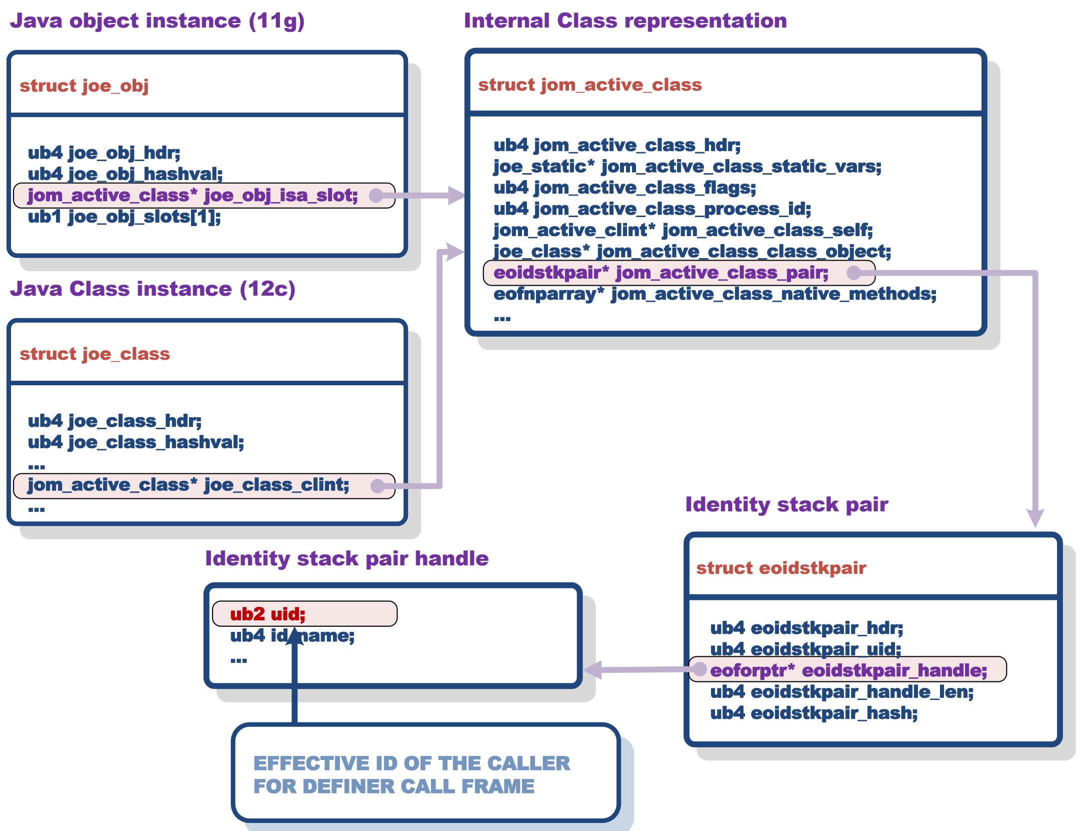
Java object instance (11g)
struct joe_obj
ub4 joe_obj_hdr;
ub4 joe_obj_hashval;
jom_active_class* joe_obj_isa_slot;
ub1 joe_obj_slots[1];
Internal Class representation
struct jom_active_class
ub4 jom_active_class_hdr;
joe_static* jom_active_class_static_vars;
ub4 jom_active_class_flags;
ub4 jom_active_class_process_id;
jom_active_clint* jom_active_class_self;
joe_class* jom_active_class_class_object;
eoidstkpair* jom_active_class_pair;
eofnparray* jom_active_class_native_methods;
...
Java Class instance (12c)
struct joe_class
ub4 joe_class_hdr;
ub4 joe_class_hashval;
...
jom_active_class* joe_class_clint;
...
Identity stack pair
struct eoidstkpair
ub4 eoidstkpair_hdr;
ub4 eoidstkpair_uid;
eoforptr* eoidstkpair_handle;
ub4 eoidstkpair_handle_len;
ub4 eoidstkpair_hash;
Identity stack pair handle
ub2 uid;
ub4 idname;
...
EFFECTIVE ID OF THE CALLER
FOR DEFINER CALL FRAME

Fig. 2 Internal memory representation of certain structures in Oracle Database Java VM.

The implementation described above lies at the origin of a definer spoofing privilege elevation technique demonstrated in our Proof of Concept Codes. By exploiting Java Reflection API implementation, one can easily break both Java type and memory safety [8]. This can be further abused to inspect internal memory representation of runtime Java objects and certain Java VM structures of a target Oracle Database. This in particular includes `jom_active_class` structure illustrated on Fig. 2.

The unpublished `eoidstkpair_handle` field can be accessed by navigating through internal Class and identity stack pair structures. It contains several fields that among other things hold a user identifier value corresponding to the owner of the class. The initial reference to internal Class structure can be read from either Java object instance (Oracle Database 11g) or Java Class instance (Oracle Database 12c).

By changing a field of `eoidstkpair_handle` structure to the SYS user id value[1], one can easily spoof the identity of the called methods and effectively the identity seen by Oracle Database security engine. There are however several important requirement for this to occur.

- The method's owner class needs to be marked as AUTHID DEFINER, so that the `jom_active_class_pair` field is assigned a non-NULL value. For classes marked as AUTHID CURRENT_USER, `jom_active_class_pair` field is usually NULL (no

[1] User identifiers can be enumerated with the use of "`select user_id || ' ' || username from dba_users order by user_id`" PL/SQL query.

identity gets pushed onto the identity stack following the invocation of class' methods, thus no need to cache `eoidstkpair` pointer value),
- The attacker needs to be granted CREATE PROCEDURE privilege in order to define Java classes in a database environment (`ALTER JAVA CLASS` construct setting `AUTHID DEFINER` property requires properly defined Java classes visible in the database environment).

Abuse of a system definer class
If CREATE PROCEDURE privilege is not available in a target environment, definer spoofing cannot be applied directly for the reasons outlined above. Privilege elevation can however still take place. All that is needed for that purpose is a system class already defined in Oracle Database environment that is marked as AUTHID DEFINER and is owned by a privileged database identity.

There are several such classes in Oracle Database environment. The `oracle.aurora.rdbms.security.DefinersVersionedObject` is in particular interesting due to the possibility to inject arbitrary, attacker provided code into the code path following the invocation of some of its methods:

```
class DefinersVersionedObject {

    DefinersVersionedObject(Schema schema, String name) {
        vo = new VersionedObject(schema, name);
    }

    ...

    void refresh() {
        vo.refresh();
    }

    Object get() {
        return vo.get();
    }
    ...
}
```

`DefinersVersionedObject` constructor initializes its `vo` field with a reference pointing to an instance of `oracle.aurora.rdbms.security.VersionedObject` class. Further, some of its methods are called from within the AUTHID DEFINER class. This creates the possibility for the abuse of the definer ID pushed as a result of the invocation of the method of `DefinersVersionedObject` class. What one needs is the possibility to set `vo` field to the object value controlled by an attacker, so that `vo.get()` method dispatch will call attacker code. This can be accomplished by the means of arbitrary memory write operation.

The object that gets written to the `vo` field does not however need to be type compatible with `VersionedObject`. It just needs to implement several dummy placeholder methods corresponding to the spoofed `oracle.aurora.rdbms.security.VersionedObject`. They are required to properly dispatch `vo.get()` method call through attacker provided code (through proper method slot). Illustration of this privilege elevation technique is shown on Fig. 3.

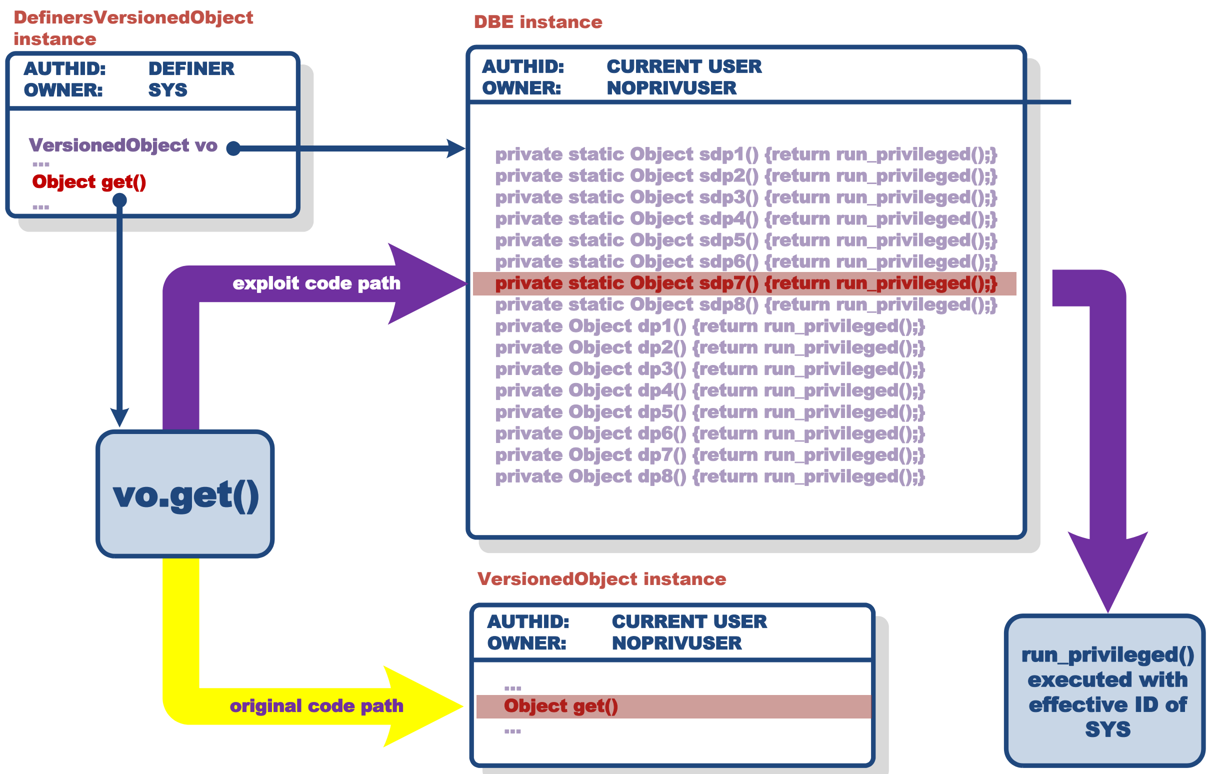
DefinersVersionedObject instance
AUTHID: DEFINER
OWNER: SYS
VersionedObject vo
...
Object get()
...
DBE instance
AUTHID: CURRENT USER
OWNER: NOPRIVUSER
private static Object sdp1() {return run_privileged();}
private static Object sdp2() {return run_privileged();}
private static Object sdp3() {return run_privileged();}
private static Object sdp4() {return run_privileged();}
private static Object sdp5() {return run_privileged();}
private static Object sdp6() {return run_privileged();}
private static Object sdp7() {return run_privileged();}
private static Object sdp8() {return run_privileged();}
private Object dp1() {return run_privileged();}
private Object dp2() {return run_privileged();}
private Object dp3() {return run_privileged();}
private Object dp4() {return run_privileged();}
private Object dp5() {return run_privileged();}
private Object dp6() {return run_privileged();}
private Object dp7() {return run_privileged();}
private Object dp8() {return run_privileged();}
exploit code path
vo.get()
original code path
VersionedObject instance
AUTHID: CURRENT USER
OWNER: NOPRIVUSER
...
Object get()
...
run_privileged() executed with effective ID of SYS

Fig. 3 Illustration of a privilege elevation technique through arbitrary definer system class.

All reported vulnerabilities and exploitation techniques were successfully verified in the environment of the following Oracle Database software:

- Oracle Database 11g Release 2 (11.2.0.1.0) for Microsoft Windows x64
- Oracle Database 11g Release 2 (11.2.0.4.5) Patch Bundle 18590877 for Microsoft Windows x64
- Oracle Database 12c Release 1 (12.1.0.1.0) for Microsoft Windows x64
- Oracle Database 12c Release 1 (12.1.0.1.9) Bundle Patch 18724015 for Microsoft Windows x64

REFERENCES

[1] Secure Coding Guidelines for the Java Programming Language, Version 4.0,
`http://www.oracle.com/technetwork/java/seccodeguide-139067.html`

[2] Security Vulnerabilities in Java SE, technical report,
`http://www.security-explorations.com/materials/se-2012-01-report.pdf`

[3] Security vulnerabilities in Oracle Java Cloud Service, Issues 1-28,
`http://www.security-explorations.com/materials/SE-2013-01-ORACLE.pdf`

[4] Oracle Database Java Developer's Guide, 11g Release 2 (11.2),
`http://docs.oracle.com/cd/E18283_01/java.112/e10588.pdf`

[5] Understanding Java Stack Inspection, Dan S. Wallach, Edward W. Felten,
`http://sip.cs.princeton.edu/pub/oakland98.pdf`

[6] David Litchfield's papers on Database Security,
`http://www.davidlitchfield.com/security.htm`

[7] Oracle Security Whitepapers from Red-Database-Security,
`http://www.red-database-security.com/whitepaper/oracle_security_whitepaper.html`

[8] Security threats in the world of digital satellite television,

http://www.security-explorations.com/materials/se-2011-01-hitb1.pdf

About Security Explorations

Security Explorations (http://www.security-explorations.com) is a security start-up company from Poland, providing various services in the area of security and vulnerability research. The company came to life in a result of a true passion of its founder for breaking security of things and analyzing software for security defects. Adam Gowdiak is the company's founder and its CEO. Adam is an experienced Java Virtual Machine hacker, with over 50 security issues uncovered in the Java technology over the recent years. He is also the hacking contest co-winner and the man who has put Microsoft Windows to its knees (vide MS03-026). He was also the first one to present successful and widespread attack against mobile Java platform in 2004.