ECG

Security Analysis Report

Sample Scan

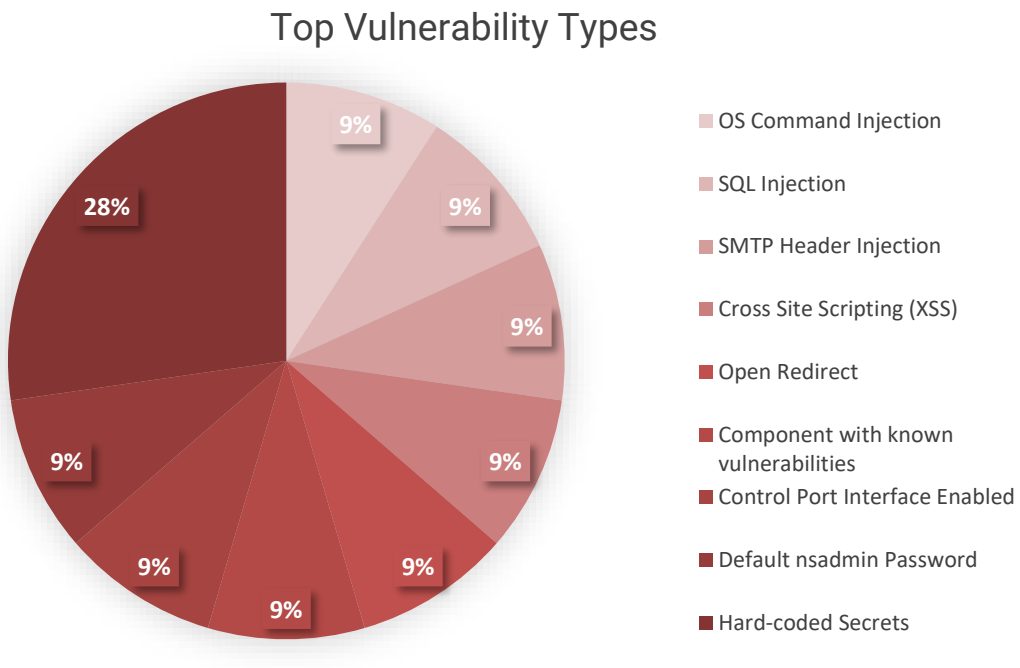
Date: 06-10-2019
User: voidsec

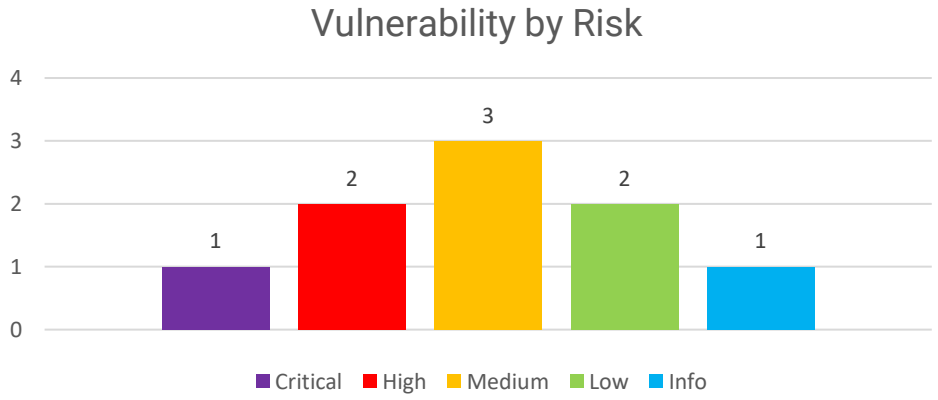
Summary

This report contains confidential information.

1. Executive Summary

Project Name:	sample scan
Analysis Start Date:	06-10-2019
Analysis End Date:	06-10-2019
Analysis Time:	0:37:21
Analysed Files:	6254
Analysed LoC:	1.279.028
LoC/min:	34.568
Files/min:	169
Scanner Settings:	clean_prj=False, debug=True, ext=None, ignore='', lang='tcl', quiet=True, scan='full', scan_level=3, target='/sample/'
Detected Issues:	9





This report contains confidential information.

2. Issues Breakdown

The detected security issues in this project are categorized as follows.

Severity	Vulnerability Type	Issues
Critical	OS Command Injection	1
High	SQL Injection	1
High	SMTP Header Injection	1
Medium	Cross Site Scripting (XSS)	1
Medium	Open Redirect	1
Medium	Component with known vulnerabilities	1
Low	Control Port Interface Enabled	1
Low	Default nsadmin Password	1
Info	Hard-coded Secrets	3

This report contains confidential information.

3. Issues Details

In the following paragraph, all security issues detected in the project codebase are presented in detail. The issues are grouped by vulnerability type.

1. OS Command Injection: exec

Severity:	Critical
Affected File:	• /sample/packages/lib/utilities.tcl:1968

exec: execute arguments as one or more shell commands

Operating system command injection vulnerabilities arise when an application incorporates user-controllable data into a command that is processed by a shell command interpreter. If the user data is not strictly validated, an attacker can use shell metacharacters to modify the command that is executed, and inject arbitrary further commands that will be executed by the server.

OS command injection vulnerabilities are usually very serious and may lead to compromise of the server hosting the application, or of the application's own data and functionality. It may also be possible to use the server as a platform for attacks against other systems. The exact potential for exploitation depends upon the security context in which the command is executed, and the privileges that this context has regarding sensitive resources on the server.

Unsanitized GET parameter “arg_list” user-supplied input reach sensitive security function on line 1968:

```
1966:    ad_proc ad_chdir_and_exec { dir arg_list } {
1967:     cd $dir
1968:     eval exec $arg_list
1969:    }
```

Remediation:

If possible, applications should avoid incorporating user-controllable data into operating system commands. In almost every situation, there are safer alternative methods of performing server-level tasks, which cannot be manipulated to perform additional commands than the one intended.

If it is considered unavoidable to incorporate user-supplied data into operating system commands, the following two layers of defence should be used to prevent attacks:

- The user data should be strictly validated. Ideally, a whitelist of specific accepted values should be used. Otherwise, only short alphanumeric strings should be accepted. Input containing any other data, including any conceivable shell metacharacter or whitespace, should be rejected.
- The application should use command APIs that launch a specific process via its name and command-line parameters, rather than passing a command string to a shell interpreter that supports command chaining and redirection

This report contains confidential information.

2. SQL Injection: ns_db dml

Severity:	High
Affected File:	• /sample/tcl/ad-referer.tcl:130

ns_db dml: executes SQL that should be data manipulation language such as an insert or update, or data definition language such as a create table.

SQL injection vulnerabilities arise when user-controllable data is incorporated into database SQL queries in an unsafe manner. An attacker can supply crafted input to break out of the data context in which their input appears and interfere with the structure of the surrounding query.

A wide range of damaging attacks can often be delivered via SQL injection, including reading or modifying critical application data, interfering with application logic, escalating privileges within the database and taking control of the database server.

Unsanitized POST parameter "Referer" user-supplied input reach sensitive security function on line 130:

```
62:     set referer [ns_set get [ns_conn headers] Referer]
105:    set foreign_url $referer
122:    set    sql_query    "select    '[ns_conn    url]',    '$foreign_url',
trunc(sysdate), 1 from dual where 0 = (select count(*) from referer_log
where local_url = '[ns_conn url]' and foreign_url = '$foreign_url'  and
trunc(entry_date) = trunc(sysdate))"
130:   ns_db dml $db $sql_query
```

Remediation:

The most effective way to prevent SQL injection attacks is to use parameterized queries (also known as prepared statements) for all database access. This method uses two steps to incorporate potentially tainted data into SQL queries: first, the application specifies the structure of the query, leaving placeholders for each item of user input; second, the application specifies the contents of each placeholder. Because the structure of the query has already been defined in the first step, it is not possible for malformed data in the second step to interfere with the query structure. You should review the documentation for your database and application platform to determine the appropriate APIs which you can use to perform parameterized queries. It is strongly recommended that you parameterize every variable data item that is incorporated into database queries, even if it is not obviously tainted, to prevent oversights occurring and avoid vulnerabilities being introduced by changes elsewhere within the code base of the application.

This report contains confidential information.

3. SMTP Header Injection: ns_sendmail

Severity:	High
Affected File:	• /sample/packages/lib/email-procs.tcl:3726

ns_sendmail: is a procedure for sending email from a Tcl script through a remote SMTP server.

SMTP header injection vulnerabilities arise when user input is placed into email headers without adequate sanitization, allowing an attacker to inject additional headers with arbitrary values. This behaviour can be exploited to send copies of emails to third parties, attach viruses, deliver phishing attacks, and often alter the content of emails. It is typically exploited by spammers looking to leverage the vulnerable company's reputation to add legitimacy to their emails.

This issue is particularly serious if the email contains sensitive information not intended for the attacker, such as a password reset token.

Unsanitized parameter "from" user-supplied input reach sensitive security function on line 3726:

```
3482:    ad_proc send_email_notification{ from } {
3726:    ns_sendmail $to $from $subject $body $extra_headers $bcc
```

Remediation:

Validate that user input conforms to a whitelist of safe characters before placing it into email headers. In particular, input containing newlines and carriage returns should be rejected. Alternatively, consider switching to an email library that automatically prevents such attacks.

This report contains confidential information.

4. Cross Site Scripting (XSS)

Severity:	Medium
Affected File:	• /sample/www/app/control/manager.tcl:1511

Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed into the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request that, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application.

The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes.

Unsanitized GET parameter "user_message" user-supplied input reach sensitive security function on line 1511:

```
1500:    set user_message [ns_set get [ns_conn form] user_msg]
1511:    ns_returnerror 500 $user_message
```

Remediation:

In most situations where user-controllable data is copied into application responses, cross-site scripting attacks can be prevented using two layers of defences:

- Input should be validated as strictly as possible on arrival, given the kind of content that it is expected to contain. For example, personal names should consist of alphabetical and a small range of typographical characters, and be relatively short; a year of birth should consist of exactly four numerals; email addresses should match a well-defined regular expression. Input which fails the validation should be rejected, not sanitized.
- User input should be HTML-encoded at any point where it is copied into application responses. All HTML metacharacters, including < > " ' and =, should be replaced with the corresponding HTML entities (< > etc).

In cases where the application's functionality allows users to author content using a restricted subset of HTML tags and attributes (for example, blog comments which allow limited formatting and linking), it is necessary to parse the supplied HTML to validate that it does not use any dangerous syntax; this is a non-trivial task.

This report contains confidential information.

5. Open Redirect: ad_returnredirect

Severity:	Medium
Affected File:	• /sample/www/admin-panel.adp:900

ad_returnredirect: write the HTTP response required to get the browser to redirect to a different page, to the current connection. This does not cause execution of the current page, including serving an ADP file, to stop.

Open redirection vulnerabilities arise when an application incorporates user-controllable data into the target of a redirection in an unsafe way. An attacker can construct a URL within the application that causes a redirection to an arbitrary external domain. This behaviour can be leveraged to facilitate phishing attacks against users of the application. The ability to use an authentic application URL, targeting the correct domain and with a valid SSL certificate (if SSL is used), lends credibility to the phishing attack because many users, even if they verify these features, will not notice the subsequent redirection to a different domain.

Unsanitized GET parameter "url" user-supplied input reach sensitive function on line 900:

```
724:    set url [ns_set get [ns_conn form] url]
900:    ad_returnredirect -allow_complete_url $url
901:    ad_script_abort
```

Remediation:

If possible, applications should avoid incorporating user-controllable data into redirection targets. In many cases, this behaviour can be avoided in the following ways:

- Remove the -allow_complete_url modifier.
- Remove the redirection function from the application, and replace links to it with direct links to the relevant target URLs.
- Maintain a server-side list of all URLs that are permitted for redirection. Instead of passing the target URL as a parameter to the redirector, pass an index into this list.

This report contains confidential information.

6. AOL <=4.5.1 log escape sequence injection

Severity:	Medium
Affected File:	• N/A

AOLServer v <=4.5.1 is vulnerable to a terminal escape injection in logs that can results in a command injection vulnerability.

Remediation:

If possible, upgrade AOL server to a version higher than 4.5.1

CVE-2009-4494 - advisory

7. Control Port Interface Enabled

Severity:	Low
Affected File:	• N/A

This control port interface allows you to telnet to a specified host and port where you can administer the server and execute database commands while the server is running.

Remediation:

If possible, disable the control port interface.

8. Default nsadmin Password

Severity:	Low
Affected File:	• N/A

By default, the nsadmin password for NaviServer is either set to NULL or to a known poor password "*CUdnvgBYocLSI*".

Remediation:

Edit the nsadmin entry in the */modules/nsperm/passwd* file. Substitute an alternate encrypted password in place of the default one.

9. Hard-coded Secrets

Severity:	Info
Affected File:	• /sample/packages/lib/api-procs.tcl:418 • /sample/packages/lib/online-procs.tcl:1518 • /sample/packages/lib/sso.tcl:1645

Hardcoded Passwords, also often referred to as Embedded Credentials, are plain text passwords or other secrets in source code. Password hardcoding refers to the practice of embedding plain text (non-encrypted) passwords and other secrets (SSH Keys, DevOps secrets, etc.) into the source code. Default, hardcoded passwords may be used across many of the same devices, applications, systems, which helps simplify set up at scale, but at the same time, poses considerable cybersecurity risk.

- /root/sample/packages/lib/api-procs.tcl:418

```
-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEA3Tz2mr7SZiAMfQyuvB
```

- /root/sample/packages/lib/online-procs.tcl:1518

```
<authentication partnerid=\""10111233\"" password=\""p4rtn3r_1233\""/>
```

- /root/sample/packages/lib/sso.tcl:1645

```
## secret key: eHxVcStBLQwA5IqzW1ir
```

Remediation:

Utilize environment variables instead of storing hard-coded secrets in the source code. Alternatively implement a Vault from HashiCorp to safely store your secrets, tokens, password, certificates, encryption keys and sensitive configurations.

This report contains confidential information.