COMMERCIAL-IN-CONFIDENCE

MEMORY SQUATTING: ATTACKS ON SYSTEM V SHARED MEMORY

VERSION: 1.0
DATE: 13/11/2013
TASK NUMBER:
System_V_Shared_Memory_Attacks_Whitepaper

PREPARED FOR

Paul Docherty
Director
Portcullis Computer Security Ltd
The Grange Barn
Pike's End
Pinner
Middlesex
HA5 2EX

Tel: +44 20 8868 0098
Email:

PREPARED BY

Tim Brown
Head Of Research
Portcullis Computer Security Limited
The Grange Barn, Pike's End
Pinner, Middlesex
HA5 2EX
United Kingdom

Tel: +44 20 8868 0098
Email: reports@portcullis-security.com

CONTENTS

Version	Author	Role	Date	Comments
0.1	TMB	Head Of Research	10/02/2013	1st Draft Whitepaper Report
0.2	TMB	Head Of Research	03/03/2013	2nd Draft Whitepaper Report - Added viability of attacks, Debian analysis, results and opportunities for memory corruption attacks
0.3	TMB	Head Of Research	04/03/2013	3rd Draft Whitepaper Report - Restructured
0.4	TMB	Head Of Research	24/05/2013	4th Draft Whitepaper Report
0.5	HJM	Editorial	08/11/2013	Review of Whitepaper Report
1.0	TMB	Head Of Research	13/11/2013	Publication

Please quote Task Number System_V_Shared_Memory_Attacks_Whitepaper in all correspondence with Portcullis.

1 Introduction

Rather than representing a definitive guide, this document represents a review of the specific security issues identified during Portcullis Computer Security Ltd's recent research into System V shared memory segments and their usage. What follows should, however, provide a high-level summary of issues, impacts and methods of remediation in cases where System V shared memory segments are used in an insecure fashion.

This paper was released as part of Tim Brown's 44CON 2013 presentation entitled "I Miss LSD".

2 Background To System V Shared Memory

System V shared memory segments are a mechanism by which applications running on POSIX alike systems can perform inter-process communications. In general terms, one process will create and write to the shared memory segment whilst other subservient processes will read from it. When more than one process requires the capability to write to a given segment, a system of semaphores is typically used to prevent inconsistencies in the segment's state.

3 Summary

System V shared memory segments created with shmget() are assigned an owner, a group and a set of permissions intended to limit access to the segment to designated processes only. The owner of a shared memory segment can change the ownership and permissions on a segment after its creation using shmctl(). Any subsequent processes that wish to attach to the segment can only do so if they have the appropriate permissions. Once attached, the process can read or write to the segment, as per the permissions that were set when the segment was created.

In the process of performing this research, we determined that a significant number of applications using shared memory segments set permissions on the segments they created that did not effectively limit access. As such, it would often be possible for other users to tamper with previously created shared memory segments or to read their contents. Moreover, similar weaknesses were identified in how applications set permissions on the semaphores intended to marshal access to these segments.

It is important to note that our research is by no means the first in this area; indeed Google fixed a similar issue (143859) in Chrome in 2012, however, no other similar research has attempted to analyse the problem in a systemic manner.

4 Patient 0: CVE-2013-0254

Qt applications such as those shipped with the KDE Software Compendium have been found to create System V shared memory segments with insecure permissions:

```
$ ipcs -a | grep 1474595
0x00000000 1474595    user        777        1024       2          dest
$ ipcs -p | grep 1474595
1474595    user          6120       6155
$ ps -aef | grep 6120
user       6120     1   0 Nov28 ?        00:02:49 /usr/bin/plasma-desktop
```

Initially it was believed that the vulnerable code was part of the KDE Software Compendium, however, by running KDE applications under a debugger as follows:

```
Breakpoint 2, shmget () at ../sysdeps/unix/syscall-template.S:82
82 in ../sysdeps/unix/syscall-template.S
(gdb) bt
bt
#0 shmget () at ../sysdeps/unix/syscall-template.S:82
#1 0x00007ffff613ac41 in ?? () from /usr/lib/x86_64-linux-gnu/libQtGui.so.4
#2 0x00007ffff6252b9f in QRasterWindowSurface::prepareBuffer(QImage::Format,
QWidget*) () from /usr/lib/x86_64-linux-gnu/libQtGui.so.4
#3 0x00007ffff6252e07 in QRasterWindowSurface::setGeometry(QRect const&) () from
/usr/lib/x86_64-linux-gnu/libQtGui.so.4
```

...it was possible to determine that the culprit was in fact Qt. Specifically, whilst it is obscured in this backtrace, the code path includes calls to QNativeImage(width, height, format, false, widget) which makes use of the following code:

```
xshminfo.shmid = shmget(IPC_PRIVATE, xshmimg->bytes_per_line * xshmimg->height, IPC_CREAT | 077
7);
```

Here, shmget() is called to implement Qt's X11 protocol support for a shared buffer between the X server and the client. This method of IPC between X clients and servers allows for increased performance when rendering large pixmaps. As you can see, in this case shmget() is called with permissions of "0777", which effectively maps to "rwx" in each of the user, group and other permissions contexts. Since the X server is typically running as root, it is believed there should be no need for client applications that run with less privileges to create shared memory segments with these weakened permissions.

Whilst performing root cause analysis of the issue above it was determined that the QSharedMemory and QSystemSemaphore classes also created shared memory segments (using shmget()) and semaphores (using semget()) with insecure permissions. Below is an example of the affected code where QSharedMemory can be seen:

```
s = new QSharedMemory("test");
s->attach();
s->create(65535, QSharedMemory::ReadOnly);
```

Executing this code was found to result in the following shared memory segment being created:

```
$ ipcs -a
...
```

```
0x51001223 3047473    user        666        65535      0
```

Note: The effect of QSharedMemory::ReadOnly is discussed later in this paper.

4.1 Analysing Debian GNU/Linux

Whilst researching the problem, we contacted the Qt security team to discuss whether the observed behaviour around Qt's usage of System V shared memory segments was desirable. Since the Qt security team were able to quickly confirm that this was not the case, we then proceeded to contact the operators of Debian Code Search to request a dump of all Debian GNU/Linux packages that call shmget(), in order to accurately determine how widespread the issue was. Armed with this list of packages, we began, naively, to analyse the various source packages with grep. This strategy proved ineffective, so we contacted the Debian and Red Hat security teams to see if they could aid us in our analysis.

Eventually, we were able to produce some simple static analysis scripts using Coccinelle, which allowed us to quickly and accurately analyse all of the C and C++ code we'd previously downloaded for insecure calls to shmget() and semget(). Coccinelle is a tool that was initially conceived almost as a semantic patcher. It understands the C language and allows you to construct generic patches for classes of bug. It has since been adopted by the Linux kernel, and a number of security researchers have began leveraging it - not just to fix bugs but also to find them. Indeed, Kees Cooke spoke at the last Linux kernel security summit on using it for exactly this.

The Coccinelle script we used in this instance is reproduced below:

```
@shmget@
expression key, size, shmflag;
position p;
@@

shmget@p(key, size, shmflag)

@script:python depends on shmget@
p << shmget.p;
shmflag << shmget.shmflag;
size << shmget.size;
@@

import re

if (re.match(".*[0-9][0-9][1-9]([\D]+.*|)$", shmflag) or re.match(".*[0-9][1-9][0-9]([\D]+.*|)$
", shmflag) or re.match(".*S_I.(GRP|OTH).*", shmflag)):
        if (re.match(".*bytes_per_line.*", size)):
                print "%s:%s: dangerous shmget(): %s (used for X)" % (p[0].file, p[0].line, shm
flag)
        else:
                print "%s:%s: dangerous shmget(): %s" % (p[0].file, p[0].line, shmflag)
elif (re.match(".*[a-z_]+[a-z_]+.*", shmflag)):
        if (re.match(".*bytes_per_line.*", size)):
                print "%s:%s: potentially dangerous shmget(): %s (used for X)" % (p[0].file, p[
0].line, shmflag)
        else:
                print "%s:%s: potentially dangerous shmget(): %s" % (p[0].file, p[0].line, shmf
lag)
```

You'll note that the script above breaks down the use of shmget() into 4 classes. Firstly, we classify whether the call to shmget() explicitly sets weak permissions when it creates shared memory segments. Secondly, we examine whether it is likely that the shared memory segment will be used to implement Qt's X11 protocol support for a shared buffer between the X server and the client (indicated by the presence of bytes_per_line in the size calculation. In cases where the call to shmget() does not explicitly set weak permissions, we further categorised those cases where the shmflag variable is set from another variable, as this can potentially be dangerous too.

4.2 Running Some Numbers

So what did our analysis show? Of the 486 Debian GNU/Linux packages that call shmget(), we found:

- 89 cases of shmget() being called insecurely for implementing X11 protocol support (58 packages)
- 212 cases of shmget() being called insecurely for other purposes (114 packages)
- 80 cases of shmget() being called potentially insecurely for implementing X11 protocol support (44 packages)
- 45 cases of semget() being called insecurely (26 packages)
- 31 cases of semget() being called potentially insecurely (23 packages)

It is important to note that of those packages that call shmget() for other purpose, some of those may actually use it for implementing X11 protocol support but do not directly reference bytes_per_line in their calls to shmget().

4.3 Viability Of Attacks

Using smaSHeM, it was possible to successfully mount a couple of attacks on the System V shared memory segments. In the context of the Qt Framework, we determined that weak permissions on the created segments may allow for the disclosure or corruption of pixmaps (GUI artifacts) being transmitted to the X server. The process by which the former was performed is described below:

1) Extract the shared memory segment you wish to view using smaSHeM as another user.
2) Call QImage(data, x, y, BITSPERPIXEL) with incrementing values of x and y.
3) Use the QImage::save() function to save each possible image.
4) Examine each of the generated images using a tool such as KDE's Gwenview to find the right dimensions of the pixmap.

It is worth noting that we did try a number of other techniques to identify the correct dimensions (such as applying an OCR and analysing the entropy of the resultant images), however, these did not prove reliable.

4.4 Opportunities For Memory Corruption Attacks

By now, some of you may be considering whether it is possible to perform memory corruption attacks against applications that utilise System V shared memory segments. To answer this question we examined how they are affected by the traditional controls that exist in POSIX alike systems to secure dynamically executed memory. We looked at two common operating systems that support shared memory segments: GNU/Linux (which of course is the base for Debian GNU/Linux) and AIX. We considered both ASLR and W^X memory corruption mitigations in this analysis.

ASLR

As it turns out, shared memory segments behave much as one might expect with respect to ASLR. On GNU/Linux, shared memory segment locations are randomised when randomize_va_space is set to 1 or higher, when a shared memory segment is attached to using shmat() whilst on AIX no randomisation of the shared memory segment location occurs. We did note, however, that it is possible to determine the location at which a shared memory segment has been created by examining /proc/<pid>/maps and looking for mappings such as "/SYSV00000000 (deleted)". Note that in the case of GNU/Linux, this presupposes that the developer hasn’t explicitly asked for the shared memory segment to be mapped to a specific fixed location with shmat().

W^X

Having looked at how ASLR affects shared memory segment behaviour, our attention then turned to W^X memory corruption mitigations. The shared memory segment permissions can be affected in two ways: Firstly, as discussed previously, an owner, a group and a set of permissions can be assigned when the segment is created with the shmget() call and, secondly, the owner of a shared memory segment can change the ownership and permissions on a segment after it has been created using the shmctl() call. In addition, the mapping can be designated as read-only when attached using the shmat() call (used to implement QSharedMemory::ReadOnly).

So how do GNU/Linux and AIX honour these various permissions?

We discovered that the permissions are honoured when assigned through calls to shmget() and shmctl() on both AIX and GNU/Linux. Attempts by us to write to shared memory segments without appropriate write permissions resulted in a segmentation fault. Similarly, attempting to write to a shared memory segment that has been mapped read-only using the shmat() call will also result in a segmentation fault. However, there is one area where behaviour is inconsistent: the way AIX and GNU/Linux enforce execute permissions that may have been assigned when the shared memory segment is created. Specifically, we observed that this permission is not honoured, either by AIX or GNU/Linux. In the case of AIX, the segment can be considered executable even when this permission has explicitly not been assigned, whilst in the case of GNU/Linux, the segment is not executable even when this permission has been explicitly assigned.

In General

Based on a cursory manual review of a selection of the packages identified earlier in this research, we found no instances of shared memory weaknesses where memory corruption was a likely outcome. In the cases where it was being utilised by the X11 protocol support for a shared buffer between the X server and the client, any such vulnerabilities would ultimately reside in the X server, whilst in other cases we typically found the memory to be accessed as a sequential list of structures of a given type. Whilst we could of course modify the contents of these using smaSHeM, no cases were identified where members of these structures were assigned with address pointers.

4.5 System V Shared Memory Segments And Keys

As described above, the System V shared memory security model is primarily based on the ownership and permissions that have been applied. Whilst performing this research, we noticed that calls to shmget() require a key parameter. From a historian’s perspective, we wondered if this was originally conceived as a security control, and it does appear that historically developers were meant to derive this key parameter from a filename using ftok(). The main page for ftok() on Debian GNU/Linux, however, indicates that:

"Of course no guarantee can be given that the resulting key_t is unique. Typically, a best effort attempt combines the given proj_id byte, the lower 16 bits of the inode number, and the lower 8 bits of the device number into a 32-bit result. Collisions may easily happen, for example between files on /dev/hda1 and files on /dev/sda1"

The presence of this information, coupled with the fact that shmat() does not require an attacker to know the key, leads us to believe that it is unlikely that key is, or ever was, a security control.

Qt’s Key Implementation

In the case of Qt, we observed that QSharedMemory implementation abstracts this operation for portability and performs ftok(filename, ’Q’). Qt creates the filename by concatenating the local temporary directory with "qipc_sharedmemory_", part of the key (a string, not to be confused with the shmget() key parameter) supplied when instantiating a QSharedMemory object and a sha1() hash of part of the very same key. For example, given a key of "test99", Qt will use a file of "/tmp/qipc_sharedmemory_test<hash of test99>" (the exact algorithm can be found in qt4-x11-4.8.2+dfsg/src/corelib/kernel/qsharedmemory.cpp). Additionally, we identified that Qt uses a similar mechanism for the construction of semget() keys.

Since both are predictable, we decided to take a look to see if there were any interesting race conditions that the implementation falls foul of. In general terms it appears that Qt's key usage is robust, however, we were able to identify that the presence of a symbolic link with the predicted filename causes the shared memory segment creation to fail in a manner that gives rise to NULL pointer dereferences, if the result of create() isn't correctly checked before attempts are then made to access the requested shared memory segment.

5 Conclusions

- System V shared memory is often created with weak permissions.
- Usage of System V shared memory by X11 applications is particularly problematic.
- Qt Project patched Qt APIs (CVE-2013-0254), Oracle patched Java JRE (CVE-2013-1500), Google patched Chrome independently.
- No progress has been made on the problem more generally by either Red Hat or Debian.
- Coccinelle is an effective tool for performing static analysis on large corpuses of C.
- Memory corruption attacks against System V shared memory are unlikely.

5.1 Future Research

Having analysed System V shared memory segments and their usage in some depth, and having looked at the implications of calling the associated functions in an insecure fashion, we would strongly recommend that other mechanisms used by POSIX alike systems to perform inter-process communications are also examined. These include:

- UNIX sockets
- System V messages (msg*)
- POSIX shared memory (shm_*) - actually covered with an example vulnerability in "I Miss LSD"
- POSIX messages (mq_*)

6 References

- http://labs.portcullis.co.uk/presentations/i-miss-lsd/
- http://codesearch.debian.net/
- http://kernsec.org/wiki/index.php/Linux_Security_Summit_2012/Abstracts/Cook
- http://coccinelle.lip6.fr/papers/stuart_thesis.pdf

7 Thanks

- Mark Lowe and Tim Varkalis of Portcullis Computer Security Ltd
- Michael Stapelberg for Debian Code Search
- Various reponders from the Red Hat, Debian, Qt and KDE security teams

Appendix A About Portcullis Computer Security Limited

Since our formation in 1986 Portcullis has developed into a widely recognized and respected provider of Information Security services with the strong foundation that comes from being an independent, mature and financially stable Company.

Portcullis' revered reputation stems from our Security Testing Service, launched back in 1996, which flourished into the professional and high quality service that our Clients benefit from today. This is further endorsed by Portcullis' array of industry accreditations and the numerous accredited CHECK Team Leaders / Members and CREST Application / Infrastructure Consultants we have, which stands testament to the investment Portcullis makes in its staff, training and R&D.

Over the years Portcullis has also expanded its key portfolio of services, which now fall into 4 main disciplines - security testing, digital forensics, cyber defence and security consultancy services. The most recent addition to our range of specialist services has been the launch of our Cyber Threat Analysis and Detection Service (CTADS®) and eDisclosure Service. These specialist IT security services not only broaden Portcullis' offering to its Clients but they also enhance and compliment each other, enabling us to deliver comprehensive solutions to our Clients as a trusted security advisor and dependable security partner.

Today, Portcullis is in the proud position of employing one of the largest multidiscipline information security resources in the UK across two locations, in Pinner (Middlesex) and Cheltenham (Gloucestershire), and has extended this capability further with international offices in San Francisco (USA) and Madrid (Spain). To accommodate the continued growth of our services and staff, we have recently commissioned a new purpose built Headquarters in Northwood that will include an HMG standards based secure facility.

With a client base encompassing Central and Local Government, Banks, Manufacturing, Charities, Telecoms, Utilities, Insurance, Retail, Healthcare, Energy, Education, Fast Moving Consumer Goods, Technology, Financial Services, Media and many international Blue Chip clients operating in EMEA and the Americas Portcullis' breadth of expertise and experience is second to none.