Exploiting large memory management vulnerabilities in Xorg server running on Linux

version 1.0

Rafal Wojtczuk
Invisible Things Lab
rafal@invisiblethingslab.com

August 17, 2010

1 Summary

A malicious authenticated client can force Xorg server to exhaust (or fragment) its address space. If running on Linux, this may result in the process stack top being in an unexpected region and execution of arbitrary code with server privileges (root). x86_32 and x86_64 platforms are affected, others most probably are affected, too.

Note that depending on the system configuration, by default local unprivileged users may be able to start an instance of Xorg server that requires no authentication and exploit it. Also if a remote attacker exploits a (unrelated) vulnerability in a GUI application (e.g. web browser), he will have ability to attack X server.

In case of a local attacker that can use *MIT-SHM* extension (which is the most likely scenario), the exploit is very reliable.

Identifier CVE-2010-2240 has been reserved for the underlying issue (Linux kernel not providing stack and heap separation). This issue has been known for at least five years.

2 Attack scenario

The class of vulnerabilities mentioned in the title is described by Gael Delalleu[1] at [1]. Applying this paper to X server case, an attacker can instruct Xorg server to allocate many large pixmaps. This may result in the stack and mmapped memory regions becoming close, and allows all the tricks described in the Gael's paper.

In fact, X server case is special, because of *MIT-SHM* extension. Local attacker can almost completely exhaust X server's address space, then create a

[1] This paper is the first summary known to the author

shared memory segment S and force X server to attach it at the only available region left, which will be close above the stack. Then attacker instructs X server to call a recursive function, which results in the stack being extended and the stack pointer being moved to S for a brief period of time (during recursion). Attacker can then write to S; this will overwrite the stack locations and allow arbitrary code execution. So, unlike in Gael's paper, we will not need to trigger data structures corruption by expanding the stack; we will be able to write to the expanded stack directly, which makes this attack 100% reliable (and somehow unique).

Without *MIT-SHM*, it is possible for an attacker to allocate a pixmap and have X server place the stack top there for a moment, but when attacker instructs X server to write to this pixmap, the stack is already in a safe location (because the recursing function has already completed).

Attack steps:

1. In case of x86_64 platform, instruct X server to allocate as many 32Kx32K pixmaps (largest allowed by X) as possible. Note that on x86_64 platform, not all 64bit address space is available - legal addresses must be canonical (top 16bits must be all 0 or all 1), so we have only 49bits address space. Linux and X have their additional restrictions on the mmap return value; as a result, ca 36000 pixmaps exhaust all address space. Note that X server does not initialize pixmap contents (just reserves a VMA for it); my tests showed that only about 800MB of RAM is needed for this step. Again, this step is not necessary in x86_32 case.

2. shm_seg_size=shmmax [2]

   ```
   while shm_seg_size >= PAGE_SIZE
       shm_seg=shmget(..., shm_seg_size,...)
       have X attach shm_seg
       if XShmAttach fails, then shm_seg_size/=2
   done
   ```

   Similarly, the action of creating and attaching a shared memory segment requires resident memory only for the control structures, not for all segment content; thus little RAM is required. On Linux, the maximum number of shared memory segments is limited by *shmmni* kernel variable (exported in */proc/sys/kernel/shmmni*). By default it is 4096, and that is why in case of a 64bits platform we need to shrink the available address space by allocating pixmaps first.

3. Allocate windows arranged so that when X processes them, some function F is called recursively. Trigger F recursion.

4. Find a shared memory segment S that has nonzero content. Nonzero content means X stack top was resident in this segment during F recursion.

[2] shmmax is the maximal shared memory segment size; from /proc/sys/kernel/shmmax

5. Spawn a process W that continuously overwrites the bottom page of S with custom payload

6. Trigger F recursion. When one of Fs returns, it will pick the return address from our payload. It is a race (W must write to stack after F has placed its return address there, but before F returns), but reliably winnable, most easily on SMP systems.

3 Workarounds

3.1 Set RLIMIT_AS

As the attack described above exhausts all address space, the natural workaround would be to set the Xorg process limit on the virtual memory (RLIMIT_AS) to TASK_SIZE-SOME_SLACK_SPACE. Then, when in allocation phase, the exploit should run out of this limit before the distance between the stack top and the allocated regions becomes less than SOME_SLACK_SPACE. One could hope that SOME_SLACK_SPACE could be rather small (around RLIMIT_STACK, around 10M).

However, attacker does not need to exhaust all the address space - she just needs to make sure that the only contiguous free space area of size *shmmax* is just below the stack top, and then attach the shared memory segment of size *shmmax*. Consider the following pattern of alloc and free:

```
x
xx
 x
 xyy
 xyyyy
 x  yy
 x  yyzzz
 x  yyzzzzzz
 x  yy   zzz
```

This way, attacker can clutter the whole address space, allocating only half of its size. More complicated alloc+free patterns are possible; currently, it is known that it is enough to allocate ca 39% of address space size in order to clutter it all, but the precise theoretical lower bound is unknown to the author.

Therefore, if this RLIMIT_AS workaround is to be used (probably by creating a wrapper around Xorg binary that sets RLIMIT_AS and executes original Xorg), one must set the limit as low as possible. In case of x86_32, it means at most 1.2G. Unfortunately, legal applications that allocate a lot of pixmaps may require more space, thus this workaround may not be suitable in all 32bits environments.

On the other hand, in x86_64 case, setting RLIMIT_AS to, say, 64G should not hurt anyone while preventing the vulnerability from being exploited.

3.2 Disable MIT-SHM

Another possibility is to disable *MIT-SHM* extension by placing the

```
Section "Extensions"
Option "MIT-SHM" "disable"
EndSection
```

directive in *xorg.conf* file. This does not prevent memory corruption, but it is believed to make it very difficult to create a successful code execution exploit. However, this impacts functionality of the server.

4 The fix

In response to prevent the described attack (and similar ones), the generic solution implemented in recent Linux kernels is to keep the top page of stack VMA unmapped; in other words, maintain a one-page gap between the stack and the rest of the areas. Note that in Gael's paper, some scenarios (e.g. usage of *alloca* with a large argument[3]) are discussed when such a protection is insufficient; but it should be enough in vast majority of cases.

The Linux kernel versions that include the commit *320b2b8de12698082609ebbc1a17165727f4c893* from Linus tree are fixed. Particularly, 2.6.35.2 and 2.6.34.4 are fixed.

5 Impact on security related software that uses Xorg server

5.1 sandbox -X

"sandbox -X" utility [2] creates an environment for executing untrusted GUI applications. For each application A, an instance of Xorg server (more precisely, Xephyr) is run; A is confined by SELinux, and can talk only to its Xephyr server. The latter connects to a "real" X server to display A's output.

Using the vulnerability described above, a malicious application can execute arbitrary code in the context of the Xephyr server, and then escalate to root by attacking the "real" Xorg server that Xephyr is allowed to talk to. As a result, the whole mechanism of "sandbox -X" is defeated.

5.2 Qubes OS

In case of Qubes [3] architecture, untrusted applications execute within a VM, and connect to a Xorg server running within this VM. Xorg server running in dom0 maps windows content from VM Xorg, and displays them on "real" display.

[3] Xorg server code does not contain code mentioned in these scenarios; particularly, it does not use *alloca*

Because dom0 X server maps pages from untrusted VMs, one could expect that its address space can be exhausted, too. However, there are two mitigating factors:

- Interaction between X server in dom0 and in VM is controlled by a *qubes_guid* process. One of the proactive safety measures implemented in *qubes_guid* since the very beginning was to detect the suspiciously high number of allocations and ask the user (in dom0) for permission to continue. Thus, the attack would require the user to manually hit the "OK" button a few thousands of times (this cannot be emulated by the exploit) in order to prepare the desired memory layout in the dom0 Xorg server.
- dom0 X server maps pages from VM read-only. Therefore, even if the user patiently allowed creation of all requested windows (see previous point), and stack has expanded into one of the mappings from VM, SIGSEGV would be delivered after first write to the stack top, and the attack would not succeed.

6 Other notes

The attack has been reproduced on Fedora 13 default install, both 32 and 64bits. Local users (say, logged in via ssh) can run "Xorg :1" and attack this process. SELinux in the enforcing mode neither prevents exploitation nor limits the executed code capabilities - it is impossible to sandbox a process that requires iopl privileges (OpenBSD privilege-separated X server could resist this attack, though).

On Fedora, Xorg executable base is not randomized, so we may happily return into locations in Xorg executable (they are at constant addresses): into execl@got in case of x86_32, or into middle of os/utils.c!System in case of x86_64. If Xorg executable base was randomized (PIE executable), its base would leak in the stack content (visible in one of the shared memory segment), so it would not help, either.

If an attacker is not local (or cannot use *MIT-SHM* extension), the attack is still possible - expanded stack may overwrite other data structures. This has not been researched, and probably would be much less reliable.

7 Timeline

- 17 June 2010 - ITL notifies X.org security team about the vulnerability
- 20 June 2010 - X.org security team suggests to discuss the issue with Linux kernel developers, as the proper solution should be implemented in the kernel
- 13 Aug 2010 - the fix is committed to Linus tree [4]
- 17 Aug 2010 - the paper is published

References

[1] Gael Delalleu, *Large memory management vulnerabilities*, http://cansecwest.com/core05/memory_vulns_delalleau.pdf

[2] Dan Walsh, *Cool things with SELinux... Introducing sandbox -X*, http://danwalsh.livejournal.com/31146.html

[3] Qubes OS project, http://www.qubes-os.org

[4] Linus Torvalds, *mm: keep a guard page below a grow-down tack segment*, http://git.kernel.org/linus/320b2b8de12698082609ebbc1a17165727f4c893