Breaking the links: Exploiting the linker

Tim Brown

December 13, 2010

<mailto:timb@nth-dimension.org.uk>
<http://www.nth-dimension.org.uk> / <http://www.machine.org.uk>

Abstract

The recent discussion relating to insecure library loading on the Microsoft Windows platform provoked a significant amount of debate as to whether GNU/Linux and UNIX variants could be vulnerable to similar attacks. Whilst the general consensus of the Slashdot herd appeared to be that this was just another example of Microsoft doing things wrong, I felt this was unfair and responded with a blog post[1] that sought to highlight an example of where POSIX style linkers get things wrong. Based on the feedback I received to that post, I decided to investigate the issue a little further. This paper is an amalgamation of what I learnt. As such it contains my own research, the discoveries of others and POSIX *lore*.

Contents

List of Tables

List of Figures

1 Technical Details

1.1 What is the linker?

The linker is a program that takes one or more objects generated by a compiler and combines them into a single executable program. On GNU/Linux and UNIX variants, linking generally consists of two stages, one during compilation and one at runtime.

1.1.1 The link editor

When a program comprises multiple object files, the link editor (often referred to as `ld`) combines these files into an executable program, resolving the dependencies as it goes along. Link editors can take objects from a collection called a library. Unless a static binary has been requested, link editors do not include the whole library in the output. Rather, they include its symbols (references from the other object files or libraries), as a guide to the runtime linker which will need to be resolved when the binary is executed.

1.1.2 The runtime linker

The runtime linker (generally known as `ld.so`) is in actuality a special loader that resolves the external dependencies (in the form of symbols) for a given executable prior to execution. It then maps access to the libraries that implement these functions in order to allow successful execution. As we will see below the way the runtime linker functions can vary significantly significantly even between ostensibly similar platforms.

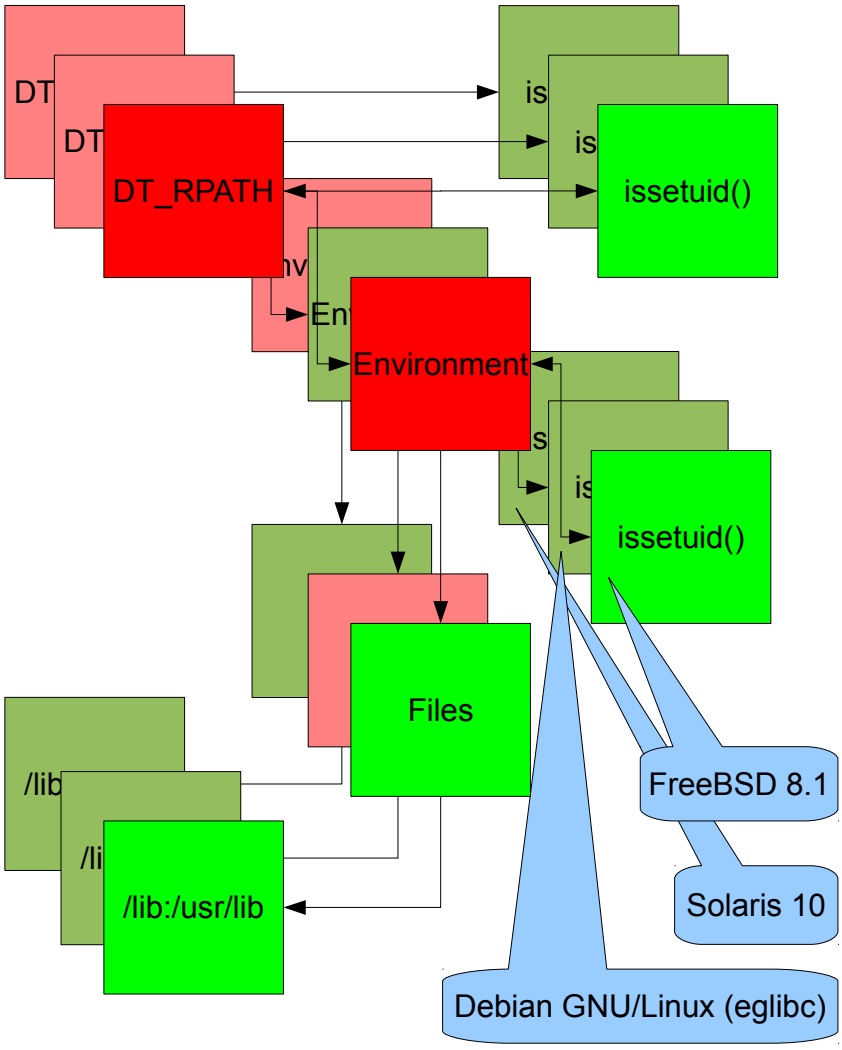
1.2 The linker attack surface

1.2.1 The process of linking and executing

In order to execute a binary, the runtime linker must resolve any dependencies to ensure that externally referenced functions are available within the excecuted binaries process space.
Firstly they look at the hints built into the binary being called. These hints take the form of the `DT_RPATH` and `DT_RUNPATH` ELF headers which consist of colon separated lists of directories that the linker should examine when looking for libraries. The linker will typically check whether the process is SetUID or not in order to determine whether macros such as `$ORIGIN` (which points at the binaries own path) should be expanded or not.
In the event that a binary does not have these ELF headers or the linker is unable to resolve all dependencies using the hinted directories, it will then examine environment variables such as `LD_LIBRARY_PATH`. As with the expansion of `$ORIGIN` care should be taken when the process is SetUID as if the linker were to trust `LD_LIBRARY_PATH` in such a case then it could be manipulated into loading malicious libraries at the behest of the calling user.
If the linker is still unable to find a dependancy, then the system's default configuration (in the form of the library cache (`ld.so.cache` on Debian GNU/Linux)) is consulted before the linker finally tries the hardcoded directories `/lib` and `/usr/lib`.
You'll notice in 1 that certain nodes have been coloured green and red. Whilst I'll explain the reasons for this in more detail later in the paper, nodes that are red contain potential weaknesses that may be exploited by manipulating various facets of the linker attack surface.

In general terms, runtime linkers implement the following flow in resolving dependencies.

Figure 1: Process flow for runtime linker



1.2.2 Environment

Linkers have a long history of being attacked via environment variables. As you can see below they vary significantly in terms of which environment variables each supports and only a small core of them are present in two or more linkers:

Table 1: Environmental attack surface for Solaris, Debian GNU/Linux and FreeBSD runtime linker

Linker Stage	Solaris 10	Debian GNU/Linux 6.0 (eglibc)	FreeBSD 8.1
ld		LD_RUN_PATH	
ld.so	LD_LIBRARY_PATH[1]	LD_LIBRARY_PATH[1]	LD_LIBRARY_PATH
	LD_PRELOAD[1,7]	LD_PRELOAD[1]	LD_PRELOAD
		LD_TRACE_LOADED_OBJECTS	LD_TRACE_LOADED_OBJECTS
	LD_BIND_NOW[7]	LD_BIND_NOW	LD_BIND_NOW
		LD_BIND_NOT	
		LD_AOUT_LIBRARY_PATH[1]	
		LD_AOUT_PRELOAD[1]	
		LD_NOWARN	
		LD_WARN	
		LD_KEEPDIR	
	LD_DEBUG[1,2,7]	LD_DEBUG[1,2]	
	LD_DEBUG_OUTPUT[2,7]	LD_DEBUG_OUTPUT[1,2]	
		LD_VERBOSE	
	LD_PROFILE[3,7]	LD_PROFILE[1,3]	
	LD_PROFILE_OUTPUT[3,7]	LD_PROFILE_OUTPUT[1,3]	
		LD_ASSUME_KERNEL	
	LD_AUDIT[7]	LD_AUDIT[4]	
	LD_CONFIG[7]		
	LD_DEMANGLE[7]		
	LD_FLAGS[7]		
	LD_LOADFLTR[7]		
	LD_NOAUDIT[7]		
	LD_NOAUXFLTR[7]		
	LD_NOCONFIG[7]		
	LD_NODIRCONFIG[7]		
	LD_NODIRECT[7]		
	LD_NOENVCONFIG[7]		
	LD_NOLAZYLOAD[7]		
	LD_NOOBJALTER[7]		
	LD_NOVERSION[7]		
	LD_ORIGIN[7]		
	LD_SIGNAL[7]		
			LD_DUMP_REL_POST
			LD_DUMP_REL_PRE
			LD_LIBMAP

Continued on next page...

Linker Stage	Solaris 10	Debian GNU/Linux 6.0 (eglibc)	FreeBSD 8.1
			LD_ LIBMAP_ DISABLE LD_ ELF_ HINTS_ PATH LD_ TRACE_ LOADED_ OBJECTS_ ALL LD_ TRACE_ LOADED_ OBJECTS_ FMT1 LD_ TRACE_ LOADED_ OBJECTS_ FMT2 LD_ TRACE_ LOADED_ OBJECTS_ PROG-NAME LD_ UTRACE

Of course there are a whole raft of other variables[2] that can affect the linker but these are the ones supported by `ld` and `ld.so` directly.

1.2.3 Files

In edition to the environment variables listed above, runtime linkers normally have default configurations which they will fall back on when the variables aren't set. The pertinant files are listed below with notes where necessary:

Solaris 10:

- `/var/ld/ld.config`
- `/var/ld/64/ld.config`

  These are typically generated using crle.

Debian GNU/Linux 6.0 (eglibc):

- `/etc/ld.so.cache`

  This is typically generated from `/etc/ld.so.conf` and updated when ldconfig is executed, for example as part of the installation of packages.

- `/etc/ld.so.preload`
- `/etc/ld.so.nohwcap`
- `/etc/suid-debug`

  Whilst this isn't documented in the man page for `ld.so` when present it changes how binaries with the SetUID bit set are executed.

[1] Cleared on SetUID/SetGID execution

[2] Writes to $0.$$

[3] Writes to /var/tmp/¡libraryname¿ or $LD_PROFILE_OUTPUT/¡libraryname¿ (Solaris) or $LD_PROFILE_OUTPUT (Debian GNU/Linux (eglibc))

[4] Exploited on glibc by Tavis Ormandy: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-CVE-2010-3847 and http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3856

[5] 32-bit versions take the form LD_32_...

[6] Exploited by Nikolaos Rangos (Kingcope): http://lists.freebsd.org/pipermail/freebsd-announce/2009-December/001289.html

[7] 32-bit versions take the form ..._32 and 64-bit versions take the form ..._64

FreeBSD 8.1:

- `/var/run/ld-elf.so.hints`
- `/var/run/ld-elf32.so.hints`
- `/etc/libmap.conf`
- `/etc/libmap32.conf`

1.2.4 issetugid() and friends

So how does the runtime linker determine whether it should trust user supplied input such the various `LD_...` environment variables. Well it depends, for the most part all runtime linkers examine the real and effective UID of the process, if these are different then the process is considered tainted and user supplied input will be ignored. That's the theory, but in practice it isn't quite that clear cut. Whilst all the linkers I've looked at make this check, it does need to be applied every time the runtime linker considers an environment variable and as Tavis pointed out, it only takes one case where the check is missed before you're in a whole world of pain. Although eglibc does tear down the `LD_...` environment variables, it only does this after it has processed them, so if there's a bug in the `ld.so` this may still be exploitable.
Another factor worth considering is the fact that the GNU/Linux world is moving away from the use of a simple SetUID bit on executable to request a change of privileges. In the last month or so, I've started to see discussions on oss-security regarding replacing the SetUID bit with file system capabilities[3]. Whilst allowing privilege changes using this mechanism should allow the privileges to be set in a far more granular manner, it will require significant changes to how processes gain and drop privileges, something we're likely to see exploited in due course.

1.3 Real world exploitation

1.3.1 The runtime linker as an interpreter

Imagine a situation in which you've by one means or another managed to get command execution as a non-privileged user and you're looking for a way to elevate your privileges to the root user. You notice that the kernel is unpatched against a known vulnerability but you can't create executable files (for example that pre-compiled version of the exploit you were playing with in your lab last week). This is a real world problem, and one that the runtime linker can help you with. You see, the runtime linker is actually an interpreter, albeit one geared for binaries:

```
user@host:~$ cp /usr/bin/id .
user@host:~$ chmod a-x id
user@host:~$ ls -la id
-rw-r--r-- 1 user user 32176 Oct 30 13:55 id
user@host:~$ ./id
bash: ./id: Permission denied
user@host:~$ /lib/ld-linux-x86-64.so.2 ./id
uid=1000(user) gid=1000(user) groups=1000(user),20(dialout),24(cdrom),25(floppy),29(audio),
44(video),46(plugdev),50(staff),116(lpadmin)
```

As you can see here, I've taken the `id` binary and removed its execute bits to simulate the case where you have write access to a file system mounted with `noexec`. Whilst it can't be run directly, the runtime linker (in this case the 64-bit version of eglibc's runtime linker) can still execute it without a problem.
So why does this work? Well, taking a look at the permissions on the runtime linker we'll start to see why:

```
lrwxrwxrwx 1 root root 12 Sep 16 12:03 /lib/ld-linux-x86-64.so.2 -> ld-2.11.2.so
-rwxr-xr-x 1 root root 128744 Sep 15 02:31 /lib/ld-2.11.2.so
```

You can see that `ld-2.11.2.so` has execute bits set. The fact is that the runtime linker is just another executable file, albeit one we rarely call directly. Indeed, on most GNU/Linux variants, the `ldd` binary is normally implemented as a shell script wrapper around it:

```
# This is the 'ldd' command, which lists what shared libraries are
# used by given dynamically-linked executables.  It works by invoking the
# run-time dynamic linker as a command and setting the environment
# variable LD_TRACE_LOADED_OBJECTS to a non-empty value.
```

From a hardening perspective, if you're mounting devices with `noexec` then you should probably ensure that they the runtime linker can't be executed either.

1.3.2 The empty library

So there's been a lot of fuss over the last couple of months about the Microsoft Insecure Library Loading Could Allow Remote Code Execution[4] vulnerability. Whilst it's fair to say that the GNU/Linux dynamic linker doesn't by default include . in its path and you'll very rarely see it listed in `ld.so.conf` and friends, there are some corner cases.
GNU/Linux and POSIX style linkers makes use of a variable called `LD_LIBRARY_PATH` which is consulted when a binary is executed and which takes precedence over the OS default as set in `ld.so.conf`. Consider the following script:

```
#!/bin/sh
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/app/lib
app start
```

What happens if `LD_LIBRARY_PATH` isn't set prior to the script being executed? Well, in that case, the `app` binary is executed with a library path of `:/path/to/app/lib`. This may seem perfectly satisfactory, but here's the rub. When the GNU/Linux dynamic linker sees a path with an empty directory specification such as `:/valid/path`, `/valid/path:` or `/valid::/path`, it treats the empty element as $PWD. This could lead to a library being loaded from the users current working directory which might it be exploitable. Go back to the shell script snippet above and consider what would happen if that was the init script for a privileged process. An administrator needs to stop and start it but he works in a security aware environment and only has access to the init script via the sudo command. So off he goes:

```
user@host:~$ sudo /etc/init.d/app
```

Sudo by default won't change your working directory when it executes a command as another user which means that `LD_LIBRARY_PATH` will end up pointing at the unprivileged user's own directory. What that means is that the GNU/Linux dynamic linker will attempt to load any library dependencies firstly from there.
Since I wrote my blog post highlighting this corner case, a number of real world examples have come to light:

- http://osvdb.org/show/osvdb/67976 - CouchDB
- http://osvdb.org/show/osvdb/68259 - SLURM
- http://osvdb.org/show/osvdb/68258 - SLURM
- http://osvdb.org/show/osvdb/68366 - Qt Creator
- http://osvdb.org/show/osvdb/68802 - TeamSpeak

So how can you set `LD_LIBRARY_PATH` safely? Well obviously you can check whether it is set before you append to it, but the following also seems to work quite nicely:

```
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH:-/path/to/app/lib}"
```

It's worth noting too that Debian (at least) are looking to fix[5] the underlying cause.

1.3.3 SIGSEGV'ing for 12 years

Whilst I was fuzzing the various runtime linkers, I came across a number of cases where I could cause my test binary to crash by manipulating the various `LD_...` environment variables. One such case was on Solaris 10 where setting `LD_PRELOAD` as follows:

```
user@host:~$ LD_PRELOAD=: su -
```

leads to a segmentation fault.
The same bug appeared to affect both SetUID and normal binaries and yields the following when when the core dump is examined with gdb:

```
Core was generated by `./test'.
Program terminated with signal 11, Segmentation fault.
#0  0xfefcfc71 in ?? ()
(gdb) x/1i $eip
0xfefcfc71:     movsbl (%esi),%ecx
(gdb) info reg esi ecx
esi            0x0      0
ecx            0x0      0
```

which is sadly is a NULL pointer dereference inside `ld.so.1`.
Further analysis showed that a similar bug was reported publicly[6] in 2005, but according to Sun this one is different. For what it's worth, it's been present since Solaris 8 and affects up to and including the last public release of OpenSolaris.
Sun have assigned issue number 7001523 to this issue.

1.3.4 What's in your RPATH?

I've already spoke in this paper on one such case where the runtime linker can be tricked into using malicious libraries using the `LD_LIBRARY_PATH` environment variable but there's actually another more interesting case which I'd like to discuss.
If you examine the linker attack surface table above you'll notice that I mention an environment variable `LD_RUN_PATH` which affects the link editor. By setting this (or indeed the `-rpath` flag) it is possible to hardset additonal locations where the runtime linker should look when resolving external

dependencies. On GNU/Linux at least, when the `DT_RPATH` or `DT_RUNPATH` exists within the ELF headers of a binary then these will be honoured first when looking for shared libraries. Additionally, the keyword `$ORIGIN` within this header is expanded to be the path of the directory where the object is found, while both `.` and the empty directory specification are honoured, even for binaries with the SetUID bit set. From an attackers pespective, SetUID binaries with `DT_RPATH` are particularly nice, since we can make use of hard links to manipulate the runtime linker into using an `$ORIGIN` which we can control.
By way of a comparison, Solaris and FreeBSD appears to ignore `$ORIGIN` for SetUID binaries and Debian patched the `$ORIGIN` issues with libc6 2.11.2-7.
Note that Solaris has another problem relating to `DT_RPATH` which I'll discuss later.

1.3.5 Debian makes me sad :(

On Debian GNU/Linux hosts, the runtime linker cache (`ld.so.cache`) is generated from the contents of `/etc/ld.so.conf.d/*` (`/etc/ld.so.conf` just includes the contents of this directory). One of the files included is `/usr/local/lib` which is writable by the `staff` group. This sounds useful but there is a problem. The library search path order generated in the cache is such that dependencies are likely to be resolved *before* the runtime linker gets as far as looking at libraries under `/usr/local/lib`. Because of this I began looking for ways to control the order in which the standard libraries are searched and in doing so I stumbled across the `LD_ASSUME_KERNEL` variable which can be set for the excution of any binary including those that have the SetUID bit set. The eglibc man page for `ld.so`[7] states that:

> Every DSO (Dynamic Shared Object, aka shared library) can tell the dynamic linker in glibc which minimum OS ABI version is needed. The information about the minimum OS ABI version is encoded in a ELF note section usually named .note.ABI-tag. This is used to determine which library to load when multiple version of the same library is installed on the system. The `LD_ASSUME_KERNEL` environment variable overrides the kernel version used by the dynamic linker to determine which library to load.

By creating a copy of `libc.so.6` under `/usr/local/lib` with an earlier ABI version and setting the `LD_ASSUME_KERNEL` environment variable to the same version, any user in the `staff` group can cause binaries with the SetUID bit set to use our copy like so:

```
user@host:~$ ./test
uid=1000,euid=0,gid=1000,egid=0
user@host:~$ LD_ASSUME_KERNEL=1.1.1 ./test
./test: error while loading shared libraries: libc.so.6: cannot open shared object file:
No such file or directory
user@host:~$ cp libc.so.6 /usr/local/lib/
user@host:~$ LD_ASSUME_KERNEL=1.1.1 ./test
uid=1000,euid=0,gid=1000,egid=0
```

It's worth noting a couple of things about the above attack. Firstly, I had to hexedit the library after compilation to change the change the value of its `.note.ABI-tag` and secondly that after copying `libc.so.6` into `/usr/local/lib`, I had to force `ldconfig` to be executed to update the runtime linker cache. In the real world you'd need to wait for an updated package to call it on your behalf during installation.
Since I orginally wrote this section of the paper, I've been doing some further research and it appears that I actually got this wrong[8], it is not required to set `LD_ASSUME_KERNEL` in order for this to be exploited as the cache generated by `ldconfig` appears to be constructed in reverse alphabetical order and therefore `/usr/local/lib` is checked before `/usr/lib`.

1.3.6 If an environment variables is set but you don't trust it, is it still there?

Having seen how eglibc's runtime linker can be manipulated, there will no doubt be a number of people cracking jokes about long haired hippies so I figured I'd take a look how FreeBSD's runtime linker compares. The FreeBSD man page for `ld.so`[9] seems to indicate that it clears the vast majority of linker related environment variables when it is used to execute but I decided to hook up my fuzzer to make sure it didn't do anything daft.
It seems that whilst it ignores the vast majority of them when resolving runtime dependencies, unlike eglibc used by Debian GNU/Linux, it doesn't actually unset[10] them in `unsetenv` which means they're inheritable by all processes spawned by the SetUID binaries. Having a look at the code responsible, it seems to use `issetugid()`[11] to determine whether to trust the environment variables. So far so good right? Well, not exactly, if the SetUID binary sets the processes real user ID based on the effective ID. This appears to untaint the running SetUID process to a degree where the runtime linker will trust the inherited linker specific environment variables including those such as `LD_PRELOAD` which can be used to modify the execution flow. Further testing appeared to show that the Solaris linker was subject to the same attack.
Whilst I haven't found any cases of SetUID binaries that are exploitable in this manner, it does show the difference that subtleties in a linker implementation can make.

1.3.7 Mapping NULL

Strange as it may seem, whilst the GNU/Linux world has finally moved to prevent userland processed from `mmap()`'ing NULL, this is not the case on Solaris where it can still be mapped. More strangely still, Sun actually provide a library which maps 0, with the following rationale (taken from the man page for `ld.so.1`[14]:

> The user compatibility library /usr/lib/0@0.so.1 provides a mechanism that establishes a value of 0 at location 0. Some applications exist that erroneously assume a null character pointer should be treated the same as a pointer to a null string. A segmentation violation occurs in these applications when a null character pointer is accessed. If this library is added to such an application at runtime using LD_PRELOAD, the library provides an environment that is sympathetic to this errant behavior. However, the user compatibility library is intended neither to enable the generation of such applications, nor to endorse this particular programming practice.
> In many cases, the presence of /usr/lib/0@0.so.1 is benign, and it can be pre-loaded into programs that do not require it. However, there are exceptions. Some applications, such as the JVM (Java Virtual Machine), require that a segmentation violation be generated from a null pointer access. Applications such as the JVM should not preload /usr/lib/0@0.so.

1.4 Auditing scripts, binaries and source

1.4.1 Scripts

To check for unsafe concatenation in shell scripts that could lead to empty directory specifications, you can create your own `libc.so` in your home directory and then wait for scripts to fail like so:

```
touch ./libc.so.6 && sudo ...
```

Whilst I'd being playing with privately, it's also fair to mention that @kees_cook also mentioned this approach on Twitter.
Additionally as described in 1.3.2, you can look for constructs such as:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/app/lib
```

1.4.2 Binaries

With binaries you should firstly check the values of any `DT_RPATH` and `DT_RUNPATH` ELF headers within the binaries using one of the following commands:

```
objdump -x ...
readelf -a ...
scanelf (from PaX)
elfdump (from Sun)
```

Secondly, as was seen in 1.3.6, you should be wary of any SetUID binaries that depend on `setuid()` and which execute further processes without unsetting any inherited environment variables.

1.4.3 Source

Finally, if you're lucky enough to have the source, keep any eye out for the following patterns which will lead to will lead to the unsafe ELF headers previously described.
You should look to identify any build scripts, Makefiles or similar which honour the `LD_RUN_PATH` environment variable.
As well as watching out for badly written build scripts you should also look at how gcc and the link editor themselves are called. The following command patters can be problematic.

```
gcc -Wl,-R,...
ld [-rpath|-rpath-link]=...
ld -R ...
```

As I've already shown with scripts, the presence of `.` or the empty directory specification of the `DT_RPATH` or `DT_RUNPATH` ELF headers, or in flags being used by `gcc` during the build process could allow libraries to be loaded from the current working directory however you should also be mindful of `$ORIGIN` macros or hard coded directory specications.

2 Changes

9th December 2010 Redacted specific vulnerabilities until vendor patches have been released, also added details of linking process

12th November 2010 Added Sun/Oracle references, Reflections on Trusting Trust revisited, Mapping NULL and other misc bug fixes

12th November 2010 Incorporated feedback including details on issetugid(), /etc/suid-debug, file system capabilities, noexec, references and other misc bug fixes. Also added details of LD_PRELOAD=: bug

8th November 2010 Initial external peer review, thanks @stealth and @taviso

References

[1] *http://www.nth-dimension.org.uk/blog.php?id=87*

[2] *http://www.scratchbox.org/documentation/general/tutorials/glibcenv.html*

[3] *http://www.openwall.com/lists/oss-security/2010/11/08/3*

[4] *http://www.microsoft.com/technet/security/advisory/2269637.mspx*

[5] *http://www.openwall.com/lists/oss-security/2010/09/29/1*

[6] *http://www.securityfocus.com/archive/1/403575/30/0/threaded*

[7] *http://manpages.ubuntu.com/manpages/lucid/man8/ldo.so.8.html*

[8] *http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=504516*

[9] *http://www.freebsd.org/cgi/man.cgi?query=ld.so*

[10] *http://svn.freebsd.org/viewvc/base/head/lib/libc/stdlib/getenv.c*

[11] *http://svn.freebsd.org/viewvc/base/head/libexec/rtld-elf/rtld.c*

[12] *http://cm.bell-labs.com/who/ken/trust.html*

[13] *http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/cmd/sgs/rtld/common/paths.c*

[14] *http://docs.sun.com/app/docs/doc/819-2239/ld.so.1-1?a=view*