Project Zero

News and updates from the Project Zero team at Google

Monday, August 25, 2014

The poisoned NUL byte, 2014 edition

Posted by Chris Evans, Exploit Writer Underling to Tavis Ormandy

Back in this 1998 post to the Bugtraq mailing list, Olaf Kirch outlined an attack he called "The poisoned NUL byte". It was an off-by-one error leading to writing a NUL byte outside the bounds of the current stack frame. On i386 systems, this would clobber the least significant byte (LSB) of the "saved %ebp", leading eventually to code execution. Back at the time, people were surprised and horrified that such a minor error and corruption could lead to the compromise of a process.

Fast forward to 2014. Well over a month ago, Tavis Ormandy of Project Zero disclosed a glibc NUL byte off-by-one overwrite into the heap. Initial reaction was skepticism about the exploitability of the bug, on account of the malloc metadata hardening in glibc. In situations like this, the Project Zero culture is to sometimes "wargame" the situation. geohot quickly coded up a challenge and we were able to gain code execution. Details are captured in our public bug. This bug contains analysis of a few different possibilities arising from an off-by-one NUL overwrite, a solution to the wargame (with comments), and of course a couple of different variants of a full exploit (with comments) for a local Linux privilege escalation.

Inspired by the success of the wargame, I decided to try and exploit a real piece of software. I chose the "pkexec" setuid binary as used by Tavis to demonstrate the bug. The goal is to attain root privilege escalation. Outside of the wargame environment, it turns out that there are a series of very onerous constraints that make exploitation hard. I did manage to get an exploit working, though, so read on to see how.

Step 1: Choose a target distribution

I decided to develop against Fedora 20, 32-bit edition. Why the 32-bit edition? I'm not going to lie: I wanted to give myself a break. I was expecting this to be pretty hard so going after the problem in the 32-bit space gives us just a few more options in our trusty exploitation toolkit.

Why Fedora and not, say, Ubuntu? Both ship pkexec by default. Amusingly, Ubuntu has deployed the fiendish mitigation called the "even path prefix length" mitigation. Kudos! More seriously, there is a malloc() that is key to the exploit, in `gconv_trans.c:__gconv_translit_find()`:

```
newp = (struct known_trans *) malloc (sizeof (struct known_trans)
                                      + (__gconv_max_path_elem_len
                                         + name_len + 3)
                                      + name_len);
```

If `__gconv_max_path_elem_len` is even, then the `malloc()` size will be odd. An odd `malloc()` size will always result in an off-by-one off the end being harmless, due to `malloc()` minimum alignment being `sizeof(void*)`.

On Fedora, `__gconv_max_path_elem_len` is odd due to the value being `/usr/lib/gconv/` (15) or `/usr/lib64/gconv/` (17). There are various unexplored avenues to try and influence this value on Ubuntu but for now we choose to proceed on Fedora.

Step 2: Bypass ASLR

Let's face it, ASLR is a headache. On Fedora 32-bit, the pkexec image, the heap and the stack are all randomized, including relative to each other, e.g.:

```
b772e000-b7733000 r-xp 00000000 fd:01 4650        /usr/bin/pkexec
b8e56000-b8e77000 rw-p 00000000 00:00 0           [heap]
bfbda000-bfbfb000 rw-p 00000000 00:00 0           [stack]
```

There is often a way to defeat ASLR, but as followers of the path of least resistance, what if we could just bypass it altogether? Well, what happens if we run pkexec again after running the shell commands `ulimit -s unlimited` and `ulimit -d 1` ? These altered limits to stack and data sizes are inherited across processes, even setuid ones:

```
40000000-40005000 r-xp 00000000 fd:01 9909        /usr/bin/pkexec
406b9000-407bb000 rw-p 00000000 00:00 0           /* mmap() heap */
bfce5000-bfd06000 rw-p 00000000 00:00 0           [stack]
```

This is much better. The pkexec image and libraries, as well as the heap, are now in static locations. The stack still moves around, with about 8MB variation (or 11 bits of entropy if you prefer), but we already know static locations for both code and data without needing to know the exact location of the stack.

(For those curious about the effect of these ulimits on 64-bit ASLR, the situation isn't as bad there. The binary locations remain well randomized. The data size trick is still very useful, though: the heap goes from a random location relative to the binary, to a static offset relative to the binary. This represents a significant reduction in entropy for some brute-force scenarios.)

Search This Blog

Search

Archives

► 2016 (17)
► 2015 (33)
▼ 2014 (11)
► December (1)
► November (2)
► October (2)
► September (1)
▼ August (2)
The poisoned NUL byte, 2014 edition
What does a pointer look like, anyway?
► July (3)

Step 3: Massage the heap using just command line arguments and the environment

After significant experimentation, our main heap massaging primitive is to call pkexec with a path comprising of '/' followed by 469 '1' characters. This path does not exist, so an error message including this path is built. The eventual error message string is a 508-byte allocation, occupying a 512-byte heap chunk on account of 4 bytes of heap metadata. The error message is built using an algorithm that starts with a 100-byte allocation. If the allocation is not large enough, it is doubled in size, plus 100 bytes, and the old allocation is freed after a suitable copy. The final allocation is shrunk to precise size using realloc. Running the full sequence through for our 508-byte string, we see the following heap API calls:

```
malloc(100), malloc(300), free(100), malloc(700), free(300), realloc(508)
```

By the time we get to this sequence, we've filled up all the heap "holes" so that these allocations occur at the end of the heap, leading to this heap layout at the end of the heap (where "m" means metadata and a red value shows where the corruption will occur):

| free space: 100 |m| free space: 300 |m| error message: 508 bytes |

In fact, the heap algorithm will have coalesced the 100 and 300 bytes of free space. Next, the program proceeds to consider character set conversion for the error message. This is where the actual NUL byte heap overflows occurs, due to our CHARSET=//AAAAA… environment variable. Leading up to this, a few small allocations outside of our control occur. That's fine; they stack up at the beginning of the coalesced free space. An allocation based on our CHARSET environment variable now occurs. We choose the number of A's in our value to cause an allocation of precisely 236 bytes, which perfectly fills the remaining space in the 400 bytes of free space. The situation now looks like this:

| blah |m| blah |m| charset derived value: 236 bytes |m: 0x00000201| error message: 508 bytes |

The off-by-one NUL byte heap corruption now occurs. It will clobber the LSB of the metadata word that precedes the error message allocation. The format of metadata is a size word, with a couple of flags in the two least significant bits. The flag 0x1, which is set, indicates that the previous buffer, the charset derived value, is in use. The size is 0x200, or 512 bytes. This size represents the 508 bytes of the following allocation plus 4 bytes of metadata. The size and flag values at this time are very specifically chosen so that the single NUL byte overflow only has the effect of clearing the 0x1 in use flag. The size is unchanged, which is important later when we need to not break forward coalescing during `free()`.

Step 4: Despair

The fireworks kick off when the error message is freed as the program exits. We have corrupted the preceding metadata to make it look like the previous heap chunk is free when in fact it is not. Since the previous chunk looks free, the malloc code attempts to coalesce it with the current chunk being freed. When a chunk is free, the last 4 bytes represent the size of the free chunk. But the chunk is not really free; so what does it contain as its last 4 bytes? Those bytes will be interpreted as a size. It turns out that as an attacker, we have zero control over these last 4 bytes: they are always 0x6f732e00, or the string ".so" preceded by a NUL byte.

Obviously, this is a very large size. And unfortunately it is used as an index backwards in memory in order to find the chunk header structure for the previous chunk. Since our heap is in the 0x40000000 range, subtracting 0x6f732e00 ends us up in the 0xd0000000 range. This address is in kernel space so when we dereference it as a chunk header structure, we get a crash and our exploitation dreams go up in smoke.

At this juncture, we consider alternate heap metadata corruption situations, in the hope we will find a situation where we have more control:

a. **Forward coalescing of free heap chunks**. If we cause the same corruption as described above, but arrange to free the chunk preceding the overflowed chunk, we follow a different code path. It results in the beginning of the 236-byte allocation being treated as a pair of freelist pointers for a linked list operation. This sounds initially promising, but again, we do not seem to have full control over the these values. In particular, the second freelist pointer comes out as NULL (guaranteed crash) and it is not immediately obvious how to overlap a non-NULL value there.
b. **Overflowing into a free chunk**. This opens up a whole range of possibilities. Unfortunately, our overflow is a NUL byte so we can only make free chunks smaller and not bigger, which is a less powerful primitive. But we can again cause confusion as to the location of heap metadata headers. See "shrink_free_hole_consolidate_backward.c" in our public bug. Again, we are frustrated because we do not have obvious control over the first bytes of any `malloc()` object that might get placed into the free chunk after we have corrupted the following length.
c. **Overflowing into a free chunk and later causing multiple pointers to point to the same memory**. This powerful technique is covered in "shrink_free_hole_alloc_overlap_consolidate_backward.c" in our public bug. I didn't investigate this path because the required precise sequence of heap operations did not seem readily possible. Also, the memory corruption occurs after the process has hit an error and is heading towards `exit()`, so taking advantage of pointers to overlapping memory will be hard.

At this stage, things are looking bad for exploitation.

Step 5: Aha! use a command-line argument spray to effect a heap spray and collide the heap into the stack

The breakthrough to escape the despair of step 4 comes when we discover a memory leak in the pkexec program; from pkexec.c:

```
else if (strcmp (argv[n], "--user") == 0 || strcmp (argv[n], "-u") == 0)
    {
      n++;
      if (n >= (guint) argc)
        {
          usage (argc, argv);
          goto out;
        }
```

```
        opt_user = g_strdup (argv[n]);
      }
```

This is very useful! If we specify multiple “-u” command line arguments, then we will spray the heap, because setting a new opt_user value does not consider freeing the old one.

Furthermore, we observe that modern Linux kernels [permit a very large number of command-line arguments](#) to be passed via `execve()`, with each one able to be up to 32 pages long.

We opt to pass a very large number (15 million+) of “-u” command line argument values, each a string of 59 bytes in length. 59 bytes plus a NUL terminator is a 60 byte allocation, which ends up being a 64 byte heap chunk when we include metadata. This number is important later.

The effect of all these command line arguments is to bloat both the stack (which grows down) and the heap (which grows up) until they crash into each other. In response to this collision, the next heap allocations actually go above the stack, in the small space between the upper address of the stack and the kernel space at 0xc0000000. We use just enough command line arguments so that we hit this collision, and allocate heap space above the stack, but do not quite run out of virtual address space -- this would halt our exploit! Once we’ve caused this condition, our tail-end mappings look a bit like this:

```
407c8000-7c7c8000 rw-p 00000000 00:00 0       /* mmap() based heap */
7c88e000-bf91c000 rw-p 00000000 00:00 0       [stack]
bf91c000-bff1c000 rw-p 00000000 00:00 0       /* another mmap() heap extent */
```

Step 6: Commandeer a malloc metadata chunk header

The heap corruption listed in step 3 now plays out in a heap extent that is past the stack. Why did we go to all this effort? Because it avoids the despair in step 4. The huge backwards index of 0x63732e00 now results in an address that is mapped! Specifically, it will hit somewhere around the 0x50700000 range, squarely in the middle of our heap spray. We control the content at this address.

At this juncture, we encounter the first non-determinism in our exploit. This is of course a shame as we deployed quite a few tricks to avoid randomness. But, by placing a heap extent past the stack, we’ve fallen victim to stack randomization. That’s one piece of randomization we were not able to bypass. By experimental determination, the top of the stack seems to range from 0xbf800000-0xbffff000, for 2048 (2^11) different possibilities with 4k (PAGE_SIZE) granularity.

Insert mode

A brief departure on exploit reliability. As we spray the heap, the heap grows in mmap() extents of size 1MB. There is no control over this. Therefore, there’s a chance that the stack will randomly get mapped sufficiently high that a 1MB mmap() heap extent cannot fit above the stack. This will cause the exploit to fail about 1 in 8 times. Since the exploit is a local privilege escalation and takes just a few seconds, you can simply re-run it.

In order to get around this randomness, we cater for every possible stack location in the exploit. The backwards index to a malloc chunk header will land at a specific offset into any one of 2048 different pages. So we simply forge a malloc chunk header at all of those locations. Whichever one hits by random, our exploit will continue in a deterministic manner by using the same path forward. At this time, it’s worth noting why we sprayed the heap with 59-byte strings. These end up spaced 64 bytes apart. Since 64 is a perfect multiple of PAGE_SIZE (4096), we end up with a very uniform heap spray pattern. This gives us two things: an easy calculation to map command line arguments to an address where the string will be placed in the heap, and a constant offset into the command line strings for where we need to place the forged heap chunk payload.

Step 7: Clobber the tls_dtor_list

So, we have now progressed to the point where we corrupt memory such that a `free()` call will end up using a faked malloc chunk header structure that we control. In order to further progress, we abuse freelist linked list operations to write a specific value to a specific address in memory. Let’s have a look at the malloc.c code to remove a pointer from a doubly-linked freelist:

```
#define unlink(AV, P, BK, FD) {                                           \
[...]
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) {                 \
    mutex_unlock(&(AV)->mutex);                                           \
    malloc_printerr (check_action, "corrupted double-linked list", P);    \
    mutex_lock(&(AV)->mutex);                                             \
  } else {                                                                \

    if (!in_smallbin_range (P->size)                                      \
        && __builtin_expect (P->fd_nextsize != NULL, 0)) {                \
      assert (P->fd_nextsize->bk_nextsize == P);                          \
      assert (P->bk_nextsize->fd_nextsize == P);                          \
      if (FD->fd_nextsize == NULL) {                                      \
[...]
      } else {                                                            \
        P->fd_nextsize->bk_nextsize = P->bk_nextsize;                     \
        P->bk_nextsize->fd_nextsize = P->fd_nextsize;                     \
[...]
```

We see that the main doubly linked list is checked in a way that makes it hard for us to write to arbitrary locations. But the special doubly linked list for larger allocations has only some debug asserts for the same type of checks. *(Aside: there’s some evidence that Ubuntu glibc builds might compile these asserts in, even for release builds. Fedora certainly does not.)* So we craft our fake malloc header structure so that the main forward and back pointers point back to itself, and so that the size is large enough to enter the secondary linked list manipulation. This bypasses the main linked list corruption check, but allows us to provide arbitrary values for the secondary linked list. These arbitrary values let us write an arbitrary 4-byte value to an arbitrary 4-byte address, but with a very significant limitation: the value we write must itself be a valid writeable address, on account of the double linking of the linked list. i.e. after we write our arbitrary value of `P->bk_nextsize` to `P->fd_nextsize`, the value `P->bk_nextsize` is itself dereferenced and written to.

This limitation does provide a headache. At this point in the process' lifetime, it is printing an error message just before it frees a few things up and exits. There are not a huge number of opportunities to gain control of code execution, and our corruption primitive does not let us directly overwrite a function pointer with another, different pointer to code. To get around this, we note that there are two important glibc static data structure pointers that indirectly control some code that gets run during the `exit()` process: `__exit_funcs` and `tls_dtor_list`. `__exit_funcs` does not work well for us because the structure contains an enum value that has to be some small number like 0x00000002 in order to be useful to us. It is hard for us to construct fake structures that contain NUL bytes in them because our building block is the NUL-terminated string. But `tls_dtor_list` is ideal for us. It is a singly linked list that runs at `exit()` time, and for every list entry, an arbitrary function pointer is called with an arbitrary value (which has to be a pointer due to previous contraints)! It's an easy version of ROP.

Step 8: Deploy a chroot() trick

For our first attempt to take control of the program, we simply call `system("/bin/bash")`. This doesn't work because this construct ends up dropping privileges. It is a bit disappointing to go to so much trouble to run arbitrary code, only to end up with a shell running at our original privilege level.

The deployed solution is to chain in a call to `chroot()` before the call to `system()`. This means that when `system()` executes `/bin/sh`, it will do so inside a chroot we have set up to contain our own `/bin/sh` program. Inside our fake `/bin/sh`, we will end up running with effective root privilege. So we switch to real root privilege by calling `setuid(0)` and then execute a real shell.

TL;DR: Done! We escalated from a normal user account to root privileges.

Step 9: Tea and medals; reflect

The main point of going to all this effort is to steer industry narrative away from quibbling about whether a given bug might be exploitable or not. In this specific instance, we took a very subtle memory corruption with poor levels of attacker control over the overflow, poor levels of attacker control over the heap state, poor levels of attacker control over important heap content and poor levels of attacker control over program flow.

Insert mode

Yet still we were able to produce a decently reliable exploit! And there's a long history of this over the evolution of exploitation: proclamations of non-exploitability that end up being neither advisable nor correct. Furthermore, arguments over exploitability burn time and energy that could be better spent protecting users by getting on with shipping fixes.

Aside from fixing the immediate glibc memory corruption issue, this investigation led to additional observations and recommendations:

- Memory leaks in setuid binaries are surprisingly dangerous because they can provide a heap spray primitive. Fixing the pkexec memory leak is recommended.
- The ability to lower ASLR strength by running setuid binaries with carefully chosen ulimits is unwanted behavior. Ideally, setuid programs would not be subject to attacker-chosen ulimit values. There's a long history of attacks along these lines, such as this recent file size limit attack. Other unresolved issues include the ability to fail specific allocations or fail specific file opens via carefully chosen `RLIMIT_AS` or `RLIMIT_NOFILE` values.
- The exploit would have been complicated significantly if the malloc main linked listed hardening was also applied to the secondary linked list for large chunks. Elevating the `assert()` to a full runtime check is recommended.
- We also noticed a few environment variables that give the attacker unnecessary options to control program behavior, e.g. G_SLICE letting the attacker control properties of memory allocation. There have been interesting historical instances where controlling such properties assisted exploitation such as this traceroute exploit from 2000. We recommend closing these newer routes too.

I hope you enjoyed this write-up as much as I enjoyed developing the exploit! There's probably a simple trick that I've missed to make a much simpler exploit. If you discover that this is indeed the case, or if you pursue a 64-bit exploit, please get in touch! For top-notch work, we'd love to feature a guest blog post.

Posted by Unknown at 6:59 PM   G+1 +29 Recommend this on Google

25 comments:

Pegasus Epsilon August 25, 2014 at 9:30 PM

Alternative to Step 8, bypassing bash's forced priv dropping when run setuid root:

http://pegasus.pimpninjas.org/code/C/sush.c

I specifically wrote this to bypass this behavior.

Reply

Replies

Tavis Ormandy August 26, 2014 at 9:58 AM

That wouldn't work, by the time your code runs privileges will already have been dropped. Just setuid(0) is enough if your code is running first, but that isn't the case here.

I suspect you might be a Debian or Ubuntu user which is a bit different, because /bin/sh is not bash. I wrote a bit about that here http://blog.cmpxchg8b.com/2013/08/security-debianisms.html.

Pegasus Epsilon February 15, 2015 at 3:42 AM

This comment has been removed by the author.

Pegasus Epsilon February 15, 2015 at 4:03 AM

I've only recently switched to Debian. I don't much care for dash, but it doesn't figure in anyway.

If you want to avoid /bin/sh -> bash dropping privs right away, use exec*() instead of system(), bypassing /bin/sh, going straight to sush, which does a setreuid() and throws you to bash with proper root privs. Or just skip all that and take the example of sush.c as intended and call setreuid (or setuid) before system, which will let you keep root. Which was the point of my post.

(If we could edit posts, the post above would still be there. Here's what I added.)

If I understand step 8 correctly, you're overwriting two atexit destructors with pointers to chroot and system to get root regardless of what /bin/sh you have on the system. Instead, you can overwrite with pointers to set(re)uid and system/exec. So system's implicit /bin/sh -c call doesn't matter at all.

Tavis Ormandy August 1, 2015 at 9:21 AM

Thanks, but you don't need to explain the difference between system and execve to us ;-)

It's being called via a corrupted tls_dtor_list, so execve does not match the required prototype, it couldn't be used here.

Reply

Nephilim August 26, 2014 at 7:35 AM

Well done. Impressive. Many thanks for sharing.

I have a question. Does this exploit work, if SELinux has been enabled (default setting)?

Reply

Insert mode

Replies

Tavis Ormandy August 26, 2014 at 9:59 AM

Yes, although I've only tested with an unconfined (default) user.

Reply

Peter Allan August 26, 2014 at 11:29 PM

A factor in this exploit is the repeated -u allowed by
sloppy argument processing. Maybe by default in getopt()
each option should be allowed only once unless indicated
otherwise.

Another use of massively repeated arguments is in my article:
http://www.zen19351.zen.co.uk/article_series/find_xargs_rm.html

Reply

Robert Larsen August 27, 2014 at 5:20 AM

/bin/bash -r
... keeps privileges if I'm not mistaken

Reply

Replies

Tavis Ormandy August 27, 2014 at 8:25 AM

You're mistaken. Bash cannot regain privileges it has already dropped, that would be a serious bug.

Think about it like this, when you do system("bash -r"), you're actually doing execve("sh -c 'bash -r'"), the second shell can't ask for the privileges back that the first shell gave up!

rnystery July 14, 2015 at 1:26 PM

And the flag is actually "-p", not "-r".

Reply

Doug McKenna August 27, 2014 at 8:33 AM

The author wrote:

... The effect of all these command line arguments is to
... bloat both the stack (which grows down) and the heap
... (which grows up) until they crash into each other.
... In response to this collision, the next heap
... allocations actually go above the stack, in the small
... space between the upper address of the stack and
... the kernel space at 0xc0000000.

Why is this not an even more serious bug? Does not every piece of code have the duty to maintain the integrity of its own data structures?

Not testing for a stack/heap collision of course means faster running code, but most of these kinds of security exploits are possible because a desire for speed and efficiency is trumping integrity/security. By allowing the stack and heap to collide, the kernel, or the design of the call stack, or whatever seems as negligent as the original off-by-1 error in glibc.

Reply

Replies

me August 27, 2014 at 3:23 PM

gael dellaleau pointed this out several years ago. sometime during the 2.6 kernel linus added a single guard page to separate stack from heap. on current linux you need to find a way to skip your pointer past the size of a page and hope that whatever your using doesn't touch the page in order to collide them.

Kyle August 28, 2014 at 5:32 PM

They don't actually overwrite each other. The heap allocations are just pushed to the other side of the stack.

Reply

marchohenle August 27, 2014 at 11:33 AM

That's why you should use musl libc http://www.musl-libc.org/
glibc is bloated garbage

I recommend the Linux distribution "Alpine Linux" that uses the musl libc

Reply

Insert mode

Hanno Böck August 27, 2014 at 2:38 PM

Just an addition, I've reported the memory leak in pkexec to the upstream devs and it's now fixed in their git code:
https://bugs.freedesktop.org/show_bug.cgi?id=83093
Also seems glibc code is now patched, too:
https://sourceware.org/bugzilla/show_bug.cgi?id=17187

Reply

Replies

taviso August 27, 2014 at 4:54 PM

Thanks Hanno!

Reply

ee9b4698-2e82-11e4-878b-57d9568b6557 August 28, 2014 at 12:14 AM

I cannot claim credit for this, it belongs, I believe, to Ilja. Its long since dead and compliments some of your supplemental points quite nicely, especially the point about ulimit's affecting setuid's.

https://www.sourceware.org/cgi-bin/cvsweb.cgi/libc/login/programs/pt_chown.c?rev=1.12&content-type=text/x-cvsweb-markup&cvsroot=glibc&only_with_tag=MAIN

"Mon May 17 18:37:12 2004 UTC (10 years, 3 months ago) by drepper"

```
[...]
static char *
more_help (int key, const char *text, void *input)
{
char *cp;

switch (key)
{
case ARGP_KEY_HELP_PRE_DOC:
asprintf (&cp, gettext ("\
Set the owner, group and access permission of the slave pseudo\
terminal corresponding to the master pseudo terminal passed on\
file descriptor `%d'. This is the helper program for the\
`grantpt' function. It is not intended to be run directly from\
the command line.\n"),
PTY_FILENO);
return cp;
case ARGP_KEY_HELP_EXTRA:
/* We print some extra information. */
asprintf (&cp, gettext ("\
The owner is set to the current user, the group is set to `%s',\
and the access permission is set to `%o'.\n\n\
%s"),
TTY_GROUP, S_IRUSR|S_IWUSR|S_IWGRP, gettext ("\
For bug reporting instructions, please see:\n\
.\n"));
return cp;
default:
break;
}
return (char *) text;
}
```

```
[...]
int
main (int argc, char *argv[])
{
[...]
setuid (getuid ());
[...]
if (remaining < argc)
{
/* We should not be called with any non-option parameters. */
error (0, 0, gettext ("too many arguments"));
argp_help (&argp, stdout, ARGP_HELP_SEE | ARGP_HELP_EXIT_ERR,
program_invocation_short_name);
[...]
```

it might be in argp_parse, i actually cant recall, but the point is the same irrelevant. unchecked asprintf() calls when run with ulimit's can fail and thus cause unitialized memory issues. setuid() can fail, again causable via ulimit's and thus two unchecked return values can be tripped by attacker controlled ulimit's causing a memory corruption issue.

Reply

Saint Crusty August 28, 2014 at 1:47 AM

Thank's for the informative and educational article as well as the educated comments.

With regards to the comment on using musl-libc, can someone care to inform about it's adoption ? Does seem like an interesting project but i'm a bit skeptic, given this is a project running in parallell to mainstream libc it will offer it's own set of limitations and imperfections ?

Reply

0xBigBan August 29, 2014 at 9:06 AM

This comment has been removed by the author.

Reply

Insert mode

g05u August 31, 2014 at 8:51 AM

tls_dtor_list not is cipher with xor?

Reply

intechsolutions srinu September 1, 2014 at 2:45 AM

Your post is really good providing good information.. I liked it and enjoyed reading it. Keep sharing such important posts.
Bulk SMS Hyderabad

Reply

Jared Gardner September 17, 2014 at 2:33 PM

PLEASE stop reporting all of the jailbreak usable bugs PLEASE

Reply

John Dudley October 9, 2014 at 2:28 AM

Acetech IT consulting Company in Delhi offer the technical knowledge and can help your business get the most out of your technology. Why are Acetech IT consulting services truly the best? Visit Us to find out more.

Reply

thedevilcorp December 30, 2014 at 8:40 AM

Good site.

Reply

Enter your comment...

Comment as: Unknown (Goog Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Insert mode