6938fa21fa047e65c7e0ca6f77d5ecf5bd2365c96e3b1e7bb5904e00e712b379

October 01, 2014

Bash bug: the other two RCEs, or how we chipped away at the original fix (CVE-2014-6277 and '78)

The patch that implements a prefix-based way to mitigate vulnerabilities in bash function exports has been out since last week and has been already picked up by most Linux vendors (plus by Apple). So, here's a quick overview of the key developments along the way, including two really interesting things: proof-of-concept test cases for two serious, previously non-public RCE bugs tracked as CVE-2014-6277 and CVE-2014-6278.

NOTE: If you or your distro maintainers have already deployed Florian's patch, there is no reason for alarm - you are almost certainly not vulnerable to attacks. If you do not have this patch, and instead relied only on the original CVE-2014-6271 fix, you probably need to act now. See this entry for a convenient test case and other tips.

Still here? Good. If you need a refresher, the basic principles of the underlying function export functionality, and the impact of the original bash bug (CVE-2014-6271), are discussed in this blog post. If you have read the earlier post, the original attack disclosed by Stephane Chazelas should be very easy to understand:

```
HTTP_COOKIE='() { 0; }; echo hi mom;' bash -c :
```

In essence, the internal parser invoked by bash to process the specially encoded function definitions passed around in environmental variables had a small problem: it continued parsing the code past the end of the function definition itself - and at that point, flat out executed whatever instructions it came across, just as it would do in a normal bash script. Given that the value of certain environmental variables can be controlled by remote attackers in quite a few common settings, this opened up a good chunk of the Internet to attacks.

The original vulnerability was reported privately and kept under embargo for roughly two weeks to develop a fairly conservative fix that modified the parser to bail out in a timely manner and do not parse any trailing commands. As soon as the embargo was lifted, we all found out about the bug and scrambled to deploy fixes. At the same time, a good chunk of the security community reacted with surprise and disbelief that bash is keen to dispatch the contents of environmental variables to a fairly complex syntax parser - so we started poking around.

Tavis was the quickest: he found that you can convince the parser to keep looking for a file name for output redirection past the boundary between the untrusted string accepted from the environment and the actual body of the program that bash is being asked to execute (CVE-2014-7169). His original test case can be simplified at:

```
HTTP_COOKIE='() { function a a>\' bash -c echo
```

This example would create an empty file named "echo", instead of executing the requested command. Tavis' finding meant that you would be at risk of remote code execution in situations where attacker-controlled environmental variables are mixed with sanitized, attacker-controlled command-line parameters passed to calls such as `system()` or `popen()`. For example, you'd be in trouble if you were doing this in a web app:

```
system("echo '"+ sanitized_string_without_quotes + "' | /some/trusted/program");
```

...because the attacker could convince bash to skip over the "echo" command and execute the command given in the second parameter, which happens to be a sanitized string (albeit probably with no ability to specify parameters). On the flip side, this is a fairly specific if not entirely exotic coding pattern - and contrary to some of the initial reports, the bug probably wasn't exploitable in a much more general way.

Chet, the maintainer of bash, started working on a fix to close this specific parsing issue, and released it soon thereafter.

On the same day, Todd Sabin and Florian Weimer have independently bumped into a static array overflow in the parser (CVE-2014-7186). The bug manifested in what seemed to be a non-exploitable crash, trying to dereference a non-attacker-controlled pointer at an address that "by design" should fall well above the end of heap - but was enough to cast even more doubt on the robustness of the underlying code. The test for this problem was pretty simple - you just needed a sequence of here-documents that overflowed a static array, say:

```
HTTP_COOKIE='() { 0 <<a <<b <<c <<d <<e <<f <<g <<h <<i <<j <<k <<l <<m; }' bash -c :
```

Florian also bumped into an off-by-one issue with loop parsing (CVE-2014-7187); the proof-of-concept function definition for this is a trivial `for` loop nested 129 levels deep, but the effect can be only observed under memory access diagnostics tools, and its practical significance is probably low. Nevertheless, all these revelations prompted him to start working on an unofficial but far more comprehensive patch that would largely shield the parser from untrusted strings in normally encountered variables present in the environment.

In parallel to Tavis' and Florian's work, I set up a very straightforward fuzzing job with american fuzzy lop. I seeded it with a rudimentary function definition:

```
() { foo() { foo; }; >bar; }
```

...and simply let it run with a minimalistic wrapper that took the test case generated by the fuzzer, put it in a variable, and then called `execve()` to invoke bash.

Although the fuzzer had no clue about the syntax of shell programs, it had the benefit of being able to identify and isolate interesting syntax based on coverage signals, deriving around 1,000 other distinctive test cases from the starting one while "instinctively" knowing not to mess with the essential "() {" prefix. For the first few hours, it kept hitting only the redirect issue originally reported by Todd and the file-creation issue discovered by Tavis - but soon thereafter, it spewed out a new crash illustrated by this snippet of code (**CVE-2014-6277**):

```
HTTP_COOKIE='() { x() { _; }; x() { _; } <<a; }' bash -c :
```

This proved to be a very straightforward use of uninitialized memory: it hit a code path in `make_redirect()` where one field in a newly-allocated `REDIR` struct - `here_doc_eof` - would not be set to any specific value, yet would be treated as a valid pointer later on (somewhere in `copy_redirect()`).

Now, if bash is compiled with both `--enable-bash-malloc` and `--enable-mem-scramble`, the memory returned to `make_redirect()` by `xmalloc()` will be set to `0xdf`, making the pointer always resolve to `0xdfdfdfdf`, and thus rendering the prospect of exploitation far more speculative (essentially depending on whether the stack or any other memory region can be grown by the attacker to overlap with this address). That said, on a good majority of Linux distros, these flags are disabled, and you can trivially get bash to dereference a pointer that is entirely within attacker's control:

```
HTTP_COOKIE="() { x() { _; }; x() { _; } <<`perl -e '{print "A"x1000}'`; }" bash -c :
bash[25662]: segfault at 41414141 ip 00190d96 sp bfbe6354 error 4 in libc-
2.12.so[110000+191000]
```

The actual fault happens because of an attempt to copy `here_doc_eof` to a newly-allocated buffer using a C macro that expands to the following code:

```
strcpy(xmalloc(1 + strlen(redirect->here_doc_eof)), (redirect->here_doc_eof))
```

This appears to be exploitable in at least one way: if `here_doc_eof` is chosen by the attacker to point in the vicinity of the current stack pointer, the apparent contents of the string - and therefore its length - may change between stack-based calls to `xmalloc()` and `strcpy()` as a natural consequence of an attempt to pass parameters and create local variables. Such a mid-macro switch will result in an out-of-bounds write to the newly-allocated memory.

A simple conceptual illustration of this attack vector would be:

```
char* result;
int len_alloced;

main(int argc, char** argv) {

  /* The offset will be system- and compiler-specific */;
  char* ptr = &ptr - 9;
  result = strcpy (malloc(100 + (len_alloced = strlen(ptr))), ptr);

  printf("requested memory = %d\n"
         "copied text = %d\n", len_alloced + 1, strlen(result) + 1);

}
```

When compiled with the -O2 flag used for bash, on one test system, this produces:

```
requested memory = 2
copied text = 28
```

Of course, the result will vary from system to system, but the general consequences of this should be fairly evident. The issue is also made worse by the fact that only relatively few distributions were building bash as a position-independent executable that could be fully protected by ASLR.

(In addition to this vector, there is also a location in `dispose_cmd.c` that calls `free()` on the pointer under some circumstances, but I haven't really really spent a lot of time trying to develop a functioning exploit for the '77 bug for reasons that should be evident in the text that follows... well, just about now.)

It has to be said that there is a bit less glamour to such a low-level issue that still requires you to go through some mental gymnastics to be exploited in a portable way. Luckily, the fuzzer kept going, and few hours later, isolated a test case that, after minimization, yielded this gem (**CVE-2014-6278**):

```
HTTP_COOKIE='() { _; } >_[$($())] { echo hi mom; id; }' bash -c :
```

I am... actually not entirely sure what happens here. A sequence of nested `$...` statements within a redirect appears to cause the parser to bail out without properly resetting its state, and puts it in the mood for executing whatever comes next. The test case works as-is with bash 4.2 and 4.3, but not with more ancient releases; this is probably related to changes introduced few years ago in bash 4.2 patch level 12 (`xparse_dolparen()`), but I have not investigated if earlier versions are patently not vulnerable or simply require different syntax.

The CVE-2014-6278 payload allows straightforward "put-your-commands-here" remote code execution on systems that are protected only with the original patch - something that we were worried about for a while, and what prompted us to ask people to update again over the past few days.

Well, that's it. I kept the technical details of the last two findings embargoed for a while to give people some time to incorporate Florian's patch and avoid the panic associated with the original bug - but at this point, given the scrutiny that the code is under, the ease of discovering the problems with off-the-shelf open-source tools, and the availability of adequate mitigations, the secrecy seems to have outlived its purpose.

Any closing thoughts? Well, I'm not sure there's a particular lesson to be learnt from the entire story. There's perhaps one thing - it would probably have been helpful if the questionable nature of the original patch was spotted by any of the notified vendors during the two-week embargo period. That said, I wasn't privy to these conversations - and hindsight is always 20/20.

11 comments:

Chris October 01, 2014 3:17 PM

Can you make any comment as to where 4.3 u28 fits into this whole situation? I was under the impression from previous comments here (and elsewhere) that 4.3 u27, posted by Chet this past Saturday after the various redhat updates, resolved all six of the currently-known Bash CVEs (including CVE-2014-7186 and CVE-2014-7187.) As such, I am surprised to see 4.3 u28 being released, especially with no accompanying updates from redhat since the 26th. Just trying to figure out how 4.3 u28 fits in and whether it specifically addresses any CVEs, since I had (perhaps incorrectly) surmised that 4.3 u27 resolved/mitigated these various CVEs being discussed.

Reply

Michal Zalewski October 01, 2014 4:58 PM

4.3.27 does not resolve all known issues, but adopts Florian's mitigation that shields the parser from untrusted inputs in normal use cases. The subsequent patch (28) actually eliminates CVE-2014-7186 and CVE-2014-7187, but with patch 27 in place, they do not pose a security risk. Two more to go, probably in patch 29.

Reply

Replies

Unknown October 02, 2014 6:04 AM

4.3.28 can resolve all 6 issues ? thanks very much

Chris October 02, 2014 6:05 AM

Thanks Michal! I assume that when you refer to Bash needing to update to resolve two more CVEs, you are referring to CVE-2014-6277 and 6278, correct?

Unknown October 02, 2014 7:48 PM

Bash 4.3.29 released on 10.2, I think this can resolve all of 6 issues, hope I am right

Reply

jul October 01, 2014 6:50 PM

if you can't be totally sure how that beast is doing, I am pretty scared.

Reply

Replies

Rich Neswold October 02, 2014 3:43 PM

Exactly. The lesson here, for me, is that I'm not getting enough bang-for-the-buck out of bash to warrant the security risks. I'll simply uninstall it from my systems.

Reply

Unknown February 21, 2015 7:53 PM

Hello everyone, just a quick question...

My impression is that scanning applies to known vulnerabilities, fuzzing is for discovering new ones, and the term "testing" can apply to both. Is that correct?

-Rick

Reply

Replies

Michal Zalewski February 21, 2015 9:27 PM

Broadly speaking, sure.

Reply

Unknown February 23, 2015 3:44 PM

...here's a very recent exploit that appears to be related to Shellshock... I just think the survivability(undetectability) and evolution of these exploits is remarkable...

https://securityblog.redhat.com/2015/02/23/samba-vulnerability-cve-2015-0240/

any thots? thanks-in-advance!

Reply

Replies

Unknown February 24, 2015 4:40 AM

apologies, in my post above, i meant to link to this article, https://securityblog.redhat.com/2014/09/24/bash-specially-crafted-environment-variables-code-injection-attack/

thanks again for any insights...

Reply

To leave a comment, click the button below to sign in with Google.

SIGN IN WITH GOOGLE

Note: Only a member of this blog may post a comment.