Emacs Arbitrary Code Execution and How to Avoid It

Details and advice about a long-standing arbitrary code execution vulnerability in Emacs

Created on [2024-11-27], last updated [2025-02-25]

This is a security advisory about CVE-2024-53920, an Emacs vulnerability that I (re-)discovered a few months ago.

TL;DR

Viewing or editing Emacs Lisp code in Emacs can run arbitrary code. The vulnerability stems from unsafe Lisp *macro-expansion*, which runs unrestricted Emacs Lisp code. Most common configurations are vulnerable (see details below). The best security measures are:

- Avoid visiting untrusted `.el` files in Emacs
- Disable automatic error checking (with Flymake or Flycheck) in untrusted `.el` files
- Disable auto-completion features in untrusted `.el` files
- UPDATE: Also set `enable-local-eval` to `nil`

This is a long-standing vulnerability which has been known for several years, but has not been addressed thus far. Emacs maintainers are working on countermeasures that will hopefully make their way into future Emacs versions. This advisory is intended to help users of existing Emacs versions protect themselves.

UPDATE: Mitigations are implemented in Emacs 30.

Update 2024-12-20

Emacs 29.4 and earlier vulnerable by default

Following the publication of this advisory, I've got a couple of emails about a way to exploit this issue in a default Emacs setup: using *file-local variables*, an attacker can bring Emacs to enable Flymake even if the user hasn't configured it to start automatically. As explained below, Flymake in Emacs versions 29.4 and earlier can execute arbitrary code when enabled in Emacs Lisp buffers.

For example, putting the following in a `.el` file and opening it in Emacs demonstrates the vulnerability in vanilla Emacs:

```
;; -*- eval: (flymake-mode 1) -*-
(rx (eval (call-process "touch" nil nil nil "/tmp/owned")))
```

To protect against this exploitation via file-local `eval` directives, set option `enable-local-eval` to `nil`.

Emacs 30 will ship with mitigations in place

The Emacs maintainers have implemented a safety mechanism which disables Flymake and code completion induced macro-expansion in untrusted files. This is already included in the latest Emacs 30 “pretest” release, version 30.0.93. See commits b5158bd1914, 8b6c6cffd1f, b9dc337ea74, 8a0c9c234f1 and 5485ea6aef9 in emacs.git for details.

Background

Macros are a staple feature across Lisp dialects. They are often cited as one of the superpowers of Lisp. They are essentially a meta-programming facility: a macro is just a Lisp function that outputs Lisp code. Since Lisp is homoiconic (code and data are represented using the same data structures), manipulating Lisp code in Lisp is as simple as processing any other program input. This makes such meta-programming fun and easy, especially in comparison to the experience of writing elaborate C preprocessor macros, for example, which often feels a bit hackish.

However, as is often the case with great powers, Lisp macros are double-edged swords—wielding them safely requires special care.

Normally, macros are executed, or “expanded”, during so-called macro-expansion time: after parsing (“reading”) text into a Lisp form, macro calls that occur in the form are expanded by executing the macro, which produces new (sub-)forms. The macro-free form obtained by expanding all macro calls can then be compiled and executed. Thus macro-expansion time comes after “read time” and before compile time and runtime.

Emacs Lisp is the programming language used implement most of Emacs’s core features and extensions, as well as to configure it. It is not the most powerful Lisp dialect out there, but it does boast a full-blown meta-programming facility in the form of macros. The problem is that macros in Emacs Lisp come with no safety measures—they can execute arbitrary, unrestricted, Emacs Lisp code. The basic macro-expansion primitive in Emacs is the Lisp function `macroexpand`, defined in C code in `src/eval.c` in the Emacs sources. It repeatedly replaces macro names with their definitions as functions, and applies those functions to the provided code:

```
while (1)
  {
    /* Come back here each time we expand a macro call,
       in case it expands into another macro call.  */
```

```
...
    {
      Lisp_Object newform = apply1 (expander, XCDR (form));
      if (EQ (form, newform))
        break;
      else
        form = newform;
    }
  }
return form;
```

That `apply1` call up there can do, well, literally anything, depending on the `expander` function (the definition of the macro) and the given input `form`.

The Emacs Lisp library `macroexp.el` provides higher-level routines on top of this `macroexpand` primitive, such as `macroexpand-all` which the Lisp byte-compiler in `bytecomp.el` uses to preprocess Lisp forms.

In addition, Emacs ships with several built-in macros that actually do execute arbitrary code by *evaluating* some of their arguments, no questions asked. These macros are `static-if`, `rx`, `cl-eval-when`, `eval-when-compile`, `eval-and-compile`, `cl-macrolet`, and perhaps others.

Therefore, if we can nudge Emacs to expand one of these macros, we get arbitrary code execution. That's the crux of this vulnerability. *Expanding macros in Emacs Lisp is unsafe by design*.

Exploitation

But could an attacker really coerce Emacs to expand macros without an explicit user request? When you open (or "visit", in Emacs parlance) an Emacs Lisp file, Emacs enables "ELisp mode", a dedicated editor mode defined in `elisp-mode.el`, which provides various useful features for exploring and editing Emacs Lisp code.

One of the features that ELisp mode provides is code completion. Completion is implemented in the function `elisp-completion-at-point`, which tries to examine the code around your cursor and come up with relevant completions. Among other things, it invokes a subroutine `elisp--local-variables` that looks for local variable names in the current scope. Since macros can completely change the meaning of the code they apply to, `elisp--local-variables` expands macros in the surrounding code to uncover local variables that may be created or obscured by such macros. Hence *invoking code completion runs arbitrary code*. In vanilla Emacs, by default, code completion is only triggered when you issue a completion command. However, since macros run arbitrary code in a Turing complete language (Emacs Lisp), there's no way to know for sure whether invoking completion will get you pwned. More importantly, almost no one uses the default Emacs configuration. Emacs users tweak various knobs, and in many common configurations folks enable auto-completion features which then trigger code completion without an explicit completion command. Such auto-completion is

performed by the popular Emacs packages Corfu and Company, as well as the newly built-in Completion Preview mode.

But the most common flow that involves automatic macro-expansion is probably *on-the-fly code diagnosis*. There are two widespread Emacs packages that check your code and warn about potential errors automatically. One is Flymake, which is built into Emacs, and the other is a popular extension package called Flycheck. Both of them, when enabled in an ELisp mode buffer, check for code issues by *byte-compiling* the code. As mentioned earlier, this involves macro-expansion, and thus arbitrary code execution. For Flymake, this byte-compilation happens in the function `elisp-flymake-byte-compile`. Like auto-completion, on-the-fly diagnosis is not enabled by default in vanilla Emacs, but it is extremely common for users to enable it. In some Emacs "distributions", such as the popular Doom Emacs and Prelude, either Flymake or Flycheck are enabled by default in ELisp mode. (UPDATE: A malicious file can use file-local variables to enable Flymake and trigger code execution in an Emacs Lisp file even in the default configuration, see example above.)

So the idea is simple: to exploit this vulnerability, an attacker crafts an Emacs Lisp file that includes a malicious macro invocation, and sends that file to an unsuspecting Emacs user. When that user opens the file in Emacs, code diagnosis is triggered automatically, which expands macros and executes arbitrary code.

Here's the content of the POC "malicious" file that I shared with the Emacs maintainers when reporting this vulnerability:

```
(rx (eval (call-process "touch" nil nil nil "/tmp/owned")))
```

If you have Flymake or Flycheck hooked to ELisp mode (again, such a setting is often the default in Emacs starter kits, and generally very common among Emacs users), then just putting the above line of code anywhere in a `.el` file and opening that file in Emacs will create a new file `/tmp/owned` on your system. Such a setup usually looks something like the following in the Emacs initialization file, `~/.emacs.d/init.el`:

```
;; Unsafe common configuration.
(add-hook 'emacs-lisp-mode-hook #'flymake-mode)
```

This is reproducible at least since Emacs version from 26.1 and all the way up to the development version of the upcoming Emacs 30.

So this is a long-standing vulnerability, and the gist of it is very simple: macros are unsafe, and in common setups Emacs expands them automatically. I've come to discover this issue while working on an enhancement for ELisp mode, which employed macro-expansion to provide semantic code highlighting. I quickly realized that doing so naively is a security risk, and soon afterwards it hit me that Emacs suffered from such a vulnerability already without my custom hacks.

The very same day, 2024-08-17, I reported my findings to the Emacs maintainers via private email. The maintainers informed me that variants of this issue have been surfaced in the past, but the issue, sadly, still stands. AFAICT the earliest public discussion about the security implications of Emacs Lisp macros started in August 2018, when Wilfred Hughes noted that code completion can lead to arbitrary code execution via macro-expansion. In October 2019, Adam Plaice reported that Flymake specifically can be used in a similar exploit. Some solutions have been floated in the discussions following these reports, but unfortunately, Emacs remains vulnerable to this very day.

(UPDATE: Emacs 30 will ship with new guardrails that are already in included in the pretest release.)

Following my report, the maintainers requested 90 days to work on a fix before public disclosure. That non-disclosure period have since expired, hence this advisory. They continue to work on a fix, which I hope will be available soon, and now we at least have a CVE to track this vulnerability. Until new guardrails are put in place to mitigate this risk, it is important to realize that macro-expansion of untrusted Emacs Lisp code is unsafe, and to be vigilant about `.el` files that you open in Emacs. Crucially, **do not enable Flymake/Flycheck in ELisp mode automatically**. Only allow automatic macro-expansion in `.el` files that you trust and control, and protect those files from tampering. (UPDATE: Also set `enable-local-eval` to `nil`, otherwise file-local `eval` directives in a malicious file can enable Flymake even if you don't enable it automatically.)

© 2023-2026 Eshel Yaron