CVE: CVE-2023-46847

Title: Squid is vulnerable to a Denial of Service

Software: Red Hat Enterprise Linux

Reporting Method: Red Hat Errata

Vulnerability: A remote attacker can perform buffer overflow attack by writing up to 2 MB of arbitrary data to heap memory when Squid is configured to accept HTTP Digest Authentication.

Note: “Squid is a high-performance proxy caching server for web clients, supporting FTP, Gopher, and HTTP data objects.” I had no idea what squid was, but picked this one because the thought of a squid being DDOS’d was highly amusing.

There are numerous reasons why buffer overflows still happen. I believe a lot of the blame can be put on C/C++ and how these languages are taught to new programmers. One of the aspects of C++ in particular is that for any specific action that a programmer would want to do, C++ offers them (I’m definitely generalizing here) about a dozen different ways to go about it. Generalizing again, of these 12 options, one of the options is the objectively best way to do it about 90% of the time. Two or three of the options have specific use cases and niches where they would outperform the main option. And all the remaining options are some combination of insecure and inefficient and should never be used in any circumstances. The main problem arises because almost all the original options to do things (like take in user input), fall into this category.

This leads to two issues. First, because they’re typically the most basic version of this action, they tend to be what is used to teach new programmers the language. Once these programmers get into the real word writing and deploying actual software, when they encounter a problem to be solved, like most people, they tend to default to what they know. The second reason is that because these functions are the oldest in the language, legacy codebases are filled with them. And to compound this issue, the amount of time and effort that would be required to purge all of these insecure functions would be immense.

Rachel

I had not considered the interactions between programming teams as something that could lead to vulnerabilities before you mentioned it, but it does make perfect sense. Assumptions that one team makes about the inputs and outputs of their sections of code may not be what other teams believe our code is assuming. I vaguely remember reading about a potential solution (or at least something that could assist) for this, making these assumptions explicit. I think the book labeled them as code or function contracts, or something like that anyways. Basically, for any function, it would be explicitly laid out in the documentation what the function takes in, and more importantly for this discussion, what the function assumes about this input. As a simple example, say there are two functions, a generic do\_stuff(), and for whatever reason, it needs to call the classic fibonacci function. Fibonacci does not like negative numbers, so the question is, which function/team's job is it to ensure that negative numbers don't break the program. Is it the do\_stuff team's job to ensure that they never pass a negative number to fibonacci, or is it the fibonacci team's job to figure out how to react when receiving a negative number? No idea what the answer should be, but the idea was that this contract should be explicitly laid out and agreed upon by these two teams, to avoid the situation where each team assumes it's the other team's responsibility.

"lets be real, nobody has flawless code from the start". Hey now, I resemble that remark!