Not even making it to the airtight hatchway: Execution even before you get there

devblogs.microsoft.com/oldnewthing/20111215-00

December 15, 2011

Raymond Chen

Today's dubious security vulnerability comes from somebody who reported that the `LoadKeyboardLayout` function had a security vulnerability which could lead to arbitrary code execution. This is a serious issue, but reading the report made us wonder if something was missing.

```
// sample program to illustrate the vulnerability.
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
int __cdecl main(int argc, char **argv)
{
 LoadKeyboardLayout("whatever", system("notepad.exe"));
 return 0;
}
```

According to the report, this sample program illustrates that the `LoadKeyboardLayout` function will execute whatever you pass as its second parameter. In this case, the program chose to launch Notepad, but obviously an attacker could change the code to something more dangerous.

We had trouble trying to figure out what the person was trying to say. After all, it's not the `LoadKeyboardLayout` function that is executing the second parameter. It's the sample program that's doing it, and using the return value as the second parameter to the `LoadKeyboardLayout` function. I mean, you can use this "technique" on the function

```
void donothing(int i) { }
```

to demonstrate that the `donothing` function has the same "vulnerability":

```
donothing(system("notepad.exe"));
```

Logically, the compiler decomposes the call to `LoadKeyboardLayout` function as

```
 auto param2 = system("notepad.exe");
 LoadKeyboardLayout("whatever", param2);
```

and now it's clear that it's not the `LoadKeyboardLayout` function which is executing its second parameter; it's *you*.

This is like taking a printed picture of your friend into a secured area, then saying, "See, I have a picture! Your security failed to stop me from taking a picture!" That picture was taken outside the secured area. What you have is not a security vulnerability because the picture was taken on the other side of the airtight hatchway.

Before contacting the submitter, we want to be sure that we weren't missing something, but after looking at it from every angle, we still couldn't see what the issue was. We ran the alleged exploit under the kernel debugger and traced through the entire `LoadKeyboard-Layout` function (both the user-mode part and the kernel-mode part) to confirm that the function never launched Notepad on its own. We repeated the investigation on all service packs on all versions of Windows still under support (and even some that are no longer supported). Still nothing.

Stumped, we contacted the submitter. "From what we can tell, the call to `system` takes place before you call the `LoadKeyboardLayout` function. Can you elaborate on how this constitutes a vulnerability in the `LoadKeyboardLayout` function?"

Apparently, the submitter didn't quite understand what we were after, because the response was just more of the same. "I have discovered that the Visual Basic `MsgBox` function has a similar vulnerability:

```
Module Program
Sub Main()
 MsgBox(System.Diagnostics.Process.Start("notepad.exe").ToString())
End Sub
End Module
```

The `MsgBox` method will execute whatever you pass as its parameter, as long as the result is a string. (You can even pass something that isn't a string, but it'll throw an exception after executing it.) The documentation for `MsgBox` clearly states that the function displays a message box with the specified text. It should therefore display a string and not execute a program!"

At this point, we had to give up. We couldn't figure out what the person was trying to report, and our attempt to obtain a clarification was met with another version of what appeared to be the same nonsense. As I recall, this entire investigation took five days to complete, plus another day or two to complete the necessary paperwork. Each year, 200,000 vulnerability reports are received, and each one is taken seriously, even the bogus-looking ones, because there might be a real issue hiding behind a bogus-looking report. Sort of how the people in the emergency communication center have to follow through on every 911 call, even the ones that they strongly suspect are bogus, and even though dealing with the suspected-bogus ones slows down the overall response time for everyone.

These sort-of-but-not-quite reports are among the most frustrating. There's enough sense in the report that it makes you wonder if there's a real vulnerability lurking in there, but which remains elusive because the author is unable (perhaps due to a language barrier) to articulate it clearly. They live in the shadowy ground between the reports that are clearly crackpot and the reports which are clear enough that you can evaluate them with confidence. These middle-ground reports are just plausible enough to be dangerous. As a result, you close them out with trepidation, because there's the risk that there really is something there, but you just aren't seeing it. Then you have nightmares that the finder has taken the report public, and the vulnerability report you rejected as bogus is now headline news all over the technology press. (Or worse, exploits start showing up taking advantage of the vulnerability you rejected as bogus two months ago.)

Update: Sure, this looks like something you can reject out of hand. But maybe there's something there after all. Perhaps the `system` call somehow "primed the pump" and left the system in just the right state so that an uninitialized variable resulted in Notepad being launched a second time or editing its token to have higher privileges. In that case, you rejected a genuine security vulnerability, and then when hackers start using it to build a botnet, somebody will go back into the vulnerability investigation logs, and the only entry will be "Rejected without investigation by Bob."

Raymond Chen

Follow