Main Menu

- Home
- Book Reviews
- Book Watch
- News
- Projects
- The Core
  - WPF Workings
  - Silverlight
  - Java
  - Joomla
  - PHP
  - JavaScript
  - C#
  - Visual Basic
  - C/C++
  - Other Languages
  - Theory
  - Artificial Intelligence
  - Mobile
  - Database
  - ASP .NET
  - Hardware
  - Graphics and Imaging
  - UI/UX
  - Perl
  - WinRT
  - HTML/CSS
  - Python
  - jQuery
  - Ruby
  - Android
- Babbage's Bag
- History
- Swift's Spreadsheets
- The Stone Tapes
- Professional Programmer
- eBooks
- Programmer Puzzles
- Bargain Computer Books
- CodeBin
- I Programmer Weekly

search...

Search

Executable Code Injection in Linux

Written by Alexey Lyashko

Monday, 26 March 2012

While code injection to patch existing software is often frowned upon, it is sometimes unavoidable. What you might not realize is that code injection is perfectly possible and often easier than you would imagine. Find out exactly how easy it is in Linux.

The common perception of the term "executable code injection" tends to shade it with malicious or threatening colors.

Although, in many cases, such perception is justified, there are at least as many where code injection is totally legal and, sometimes, even unavoidable. Imagine a situation – you have a piece of legacy software in your possession, but, for some reasons, you do not have any of its sources. I bet, some of you would now say: "Nah, what kind of developer you are if you loose your sources!". While I have been lucky not to loose any of the sources I may need so far, such things happen. Sometimes, they happen in even relatively big companies.

So, for the sake of the example, let us imagine, that you have no source code for the that piece of software. It works alright until one day you update your system and discover, that the previously brilliant software is not working as expected any more as one of its procedures utilizes resources which are no longer available.

What are you going to do about it?

On one hand, you can always (well, almost always) modify the binary by overwriting part of its code with new code. But what happens if the original binary does not have enough space to contain your new code without a risk of having dramatic impact on the functionality of the executable (e.g. corrupting offsets, etc.) There are many difficulties with attempting to statically patch the code in this way.

The alternative is to do the job dynamically. You can always create a tiny loader, which would load the software, inject modified code (even if it needs to allocate a separate block of memory for it) and perform all of the actions needed to redirect the execution flow to that code instead of the malfunctioning code.

As another example, what if you have to update a running service, which you cannot stop. Code injection is then what may help you resolve the problem.

Code injection?
What are you talking about?

It may sound odd or even unbelievable to you, but there are many programmers who think that code injection is either too complicated or even impossible.

I refer of course mainly to the world of Windows but there may be many in the world of Linux who think this way. Kris Kaspersky says there are even generations of such programmers. However, this technique, is one of the easiest (at least, as long as it is not associated with malicious intent) regardless of the operating system (Windows or Linux) and this is what I am going to cover in this article.

The Mighty *ptrace()*

Linux does not provide us with too many ways to mess around with a running executable. However, the most popular one is quite powerful.

Although, the *ptrace()* man page suggests that similar functionality is much better implemented on other systems and that it is highly OS- and architecture-specific, it usually provides us with all the needed functionality.

New Articles!

- Exploring Edison - Pulse Width Modulation
- Introduction to Cashflow
- Nuanced jQuery Versioning: Which to Choose
- Exploring Edison - Almost Real Time Linux
- Effective Interest Rates
- Getting Started With jQuery - Advanced Ajax To The Client
- Android Adventures - ListView And Adapters
- Ever Increasing Need For Secure Programming
- Simple and Compound Interest - Time Is Money
- Getting Started With jQuery - Advanced Ajax
- Useful Windows

This function is declared in *sys/ptrace.h* header file and has the following prototype:

```
long ptrace(
     enum __ptrace_request request,
     pid_t pid,
     void *addr,
     void *data);
```

request – determines the action to be performed. We will only be interested in a few of the many.

pid – the ID of the process which the desired action should be performed on.

addr – pointer to or offset into a structure within or related to the process being traced.

data – pointer to a variable or structure that would either receive data from traced process, or passed to it.

"*The ptrace() system call provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing.*" - ptrace(2) man page.

Although, parameters of this function are processed according to the prototype, it is considered to be a variadic function in modern *glibc*. However, using it as *variadic* is considered to be the utilization of undocumented features. Therefore, it is highly recommended that you read the *ptrace()* man page on your system prior to any attempt to implement the technique described below.

Spawn or Attach

Attach to process

As you may conclude from the above paragraph, this function is, mostly, used to debug an executable and as we know – there are two ways of initiating debugging:

- Attach to running process.
- Initiate the target process from within a debugger.

ptrace() allows both approaches, however, there may be some limitations.

For example, If you are using Ubuntu 10.10+, then you can only attach to a child process, meaning that the target process has to be a descendant of the one performing the debug operations.

Either way, if your system allows attaching to a non-descendant process, you may do so by calling *ptrace()*:

```
ptrace(
   PTRACE_ATTACH,
   target_process_id,
   NULL,
   NULL);
```

PTRACE_ATTACH – action to be performed (attach to process);

target_process_id – ID of the process to attach to;

addr and *data* are set to NULL as they are not important in this case.

In general, the process you are attaching to should stop awaiting for "further instructions" immediately upon attach, but it is not necessarily going to happen and such a stop may not be immediate.

Spawn a process

On the other hand, if there is a need for a permanent change (usually, this is the case) in the execution flow, it would be a better choice to create a loader for the target program.

This is exactly what I am going to describe in the rest of this article.

fork() → exec() → ptrace()

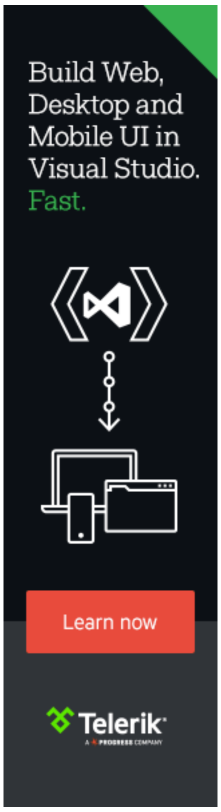
Linux kernel provides several system calls for creation of a new process:

sys_clone – creates a copy of the running process with or without shared resources (memory, file descriptors, etc.);

sys_execve – replaces the running process with a new one (has several variations in the C library);

fork - creates a copy of the running process but without any shared resources (Actually, both *sys_fork* and *sys_clone* come down to *do_fork()* function in the kernel).

We are particularly interested in *sys_fork* and *sys_execve.* Well, not in the system calls themselves, but in the *libc*'s



Developing with AngularJS?
Save time with 70+ HTML5 UI widgets Built with AngularJS In mind
Learn more
Telerik

• Simply Javascript

wrappers – *fork()* and *execl()*. In fact, *fork()* may be the first function called by our loader, unless you decide to load the to-be-injected code prior to forking the process.

The following is a skeleton for the loader:

```
/*
Function execl() has this prototype:
int execl(conts char* path,
                 const char* arg, … );
which means it is a variadic function.
path – path to the executable file
                   (may be relative)
arg – command line arguments to
be passed to the newly created process
(must be terminated by a NULL pointer)
*/
int main(int argc, char** argv)
{
 pid_t pid;
 if(0 == (pid = fork()))
 {
  // We are in the child process,
  // so we load the target executable
  ptrace(PTRACE_TRACEME, 0, NULL, NULL);
  execl(*(argv+1), NULL, NULL);
 }
 else
 {
  //Here comes the code of the injector
  //(pid = process ID of the child)
 }
 return 0;
}
```

Prev - Next >>

Last Updated (Monday, 26 March 2012)

RSS feed of all content

RSS 2.0

DETECT & PROTECT
Learn More About
DDoS Protection Solutions
AdChoices
ARBOR
NETWORKS
GET THE REPORT>