http://hick.org/code/skape/papers/needle.txt


```
[-------------------------------------------------------------------------]
[-- Uninformed Research -- informative information for the uninformed. --]
[-------------------------------------------------------------------------]
[-- Genre  : Development                                                --]
[-- Name   : needle                                                     --]
[-- Desc   : Linux x86 run-time process manipulation                    --]
[-- Url    : http://www.uninformed.org/                                 --]
[-- Use    : EVILNESS                                                   --]
[-------------------------------------------------------------------------]
[-- Author : skape (mmiller@hick.org)                                  --]
[-- Date   : 01/19/2003                                                 --]
[-------------------------------------------------------------------------]

[-- Table of contents:                                                  --]

    1) Overview
       1.1) Topics
       1.2) Techniques
       1.3) Execution Diversion
    2) Memory Allocation
    3) Memory Management
    4) Library Injection
    5) Code Injection
       5.1) Forking
       5.2) Threading
       5.3) Function Trampolines
    6) Conclusion
    7) References

[-- 1) Overview                                                         --]

    So, you want to be evil and modify the image of an executing
    process?  Well, perhaps you've come to the right place.  This
    document deals strictly with some methodologies used to to
    alter process images under Linux.  If you're curious about how
    to do something similar to the things listed in this document in
    Windows, please read the ``References`` section.

[-- 1.1) Topics                                                         --]
```

```
The following concepts will be discussed in this document as they
relate to run-time process manipulation:

* Memory Allocation

   The use of being able to allocate and deallocate memory in a
   running process from another process has awesome power for
   such scenarios as execution diversion (the act of diverting
   a processes execution to your own code), data hiding (the act
   of hiding data in a process image), and even, in some cases
   allocating dynamic structures/strings for use within a process
   for its normal execution.  These aren't the only uses, but
   they're all I could think of right now :).  See the
   ``Memory Allocation`` section for details.

* Memory Management

   The ability to copy arbitrary memory from one process to another
   at arbitrary addresses allows for flexible manipulation of
   a given processes memory image.  This can be applied to copy
   strings, functions, integers, everything.  See the ``Memory
   Management`` section for details.

* Library Injection

   The ability to inject arbitrary shared objects into a process
   allows for getting at symbols that an executable would not
   normally have as well as allowing an evil-doer such as yourself
   to inject arbitrary PIC that can reference symbols in
   an executable without getting in trouble.  This alone is
   extremely powerful.  See the ``Library Injection`` section for
   details.

* Code Injection

   Well, when you get down to it, you just want to execute code
   in a given process that you define and you want to control
   when it gets executed.  Lucky for you, this is possible AND
   just as powerful as you'd hoped.  This document will cover
   three types of code injection:

   1) Forking

      The act of causing a process to create a child image
      and execute arbitrary code.

   2) Threading
```

```
          The act of causing a process to create a thread
          that executes an arbitrary function.

       3) Function Trampolines

          The act of causing a call to a given function to
          'trampoline' to arbitrary code and then 'jump' back to
          the original function.

[-- 1.2) Techniques                                                --]

    As of this document I'm aware of two plausible techniques for
    altering the image of an executing process:

    * ptrace

       Likely the most obvious technique, the ptrace (process trace) API
       allows for altering of memory, reading of memory, looking and
       setting registers, as well as single-stepping through a process.
       The application for these things as it pertains to this document
       should be obvious.  If not, or if you're curious, read the
       ``References`` section for more details on ptrace.

    * /proc/[pid]/mem

       This technique is more limited in the amount of things it can
       do but is by no means something that should be cast aside.
       With the ability to read/write a given process's image, one
       could easily modify the image to do ``Code Injection``.  Doing
       things like memory allocation, management, and library
       injection via this method are quote a means harder but *NOT*
       impossible.  They would take a decent amount of hackery though.
       (Theoretical, not proven yet, by me at least.)

[-- 1.3) Execution Diversion                                       --]

    In order to do most of the techniques in this document we need to
    divert the execution of a running process to code that we control.
    This presents a few problems off the bat.  Where can we safely put
    the code that we want executed?  How could we possibly change the
    course of execution?  How do we restore execution once our code
    has finished?  Well, thankfully, there are answers to these
    questions, and they're pretty easy to answer.  Let's start with
    the first one.

    * Where can we safely put the code that we want executed?
```

Well to answer this question you need to have a slight understanding of how the process is laid out and how the flow of execution goes. The basic tools you need in your knowledge base are that executables have symbols, symbols map to vma's that are used to tell the vm where symbols should be located in memory. This is used not only for functions, but also for global variables. With that said, we can tell where code will be in an executable based off processing the ELF image associated with the process. Example:

```
root@rd-linux:~# objdump --syms ./ownme | grep main
08048450 g     F .text  00000082              main
```

This tells us that main will be found at 0x08048450 when the program is executing. But what good does this do us? A lot. Considering the main function is the 'gateway' to normal code execution, it's an excellent place to use as a dumping zone for arbitrary code. There are some restrictions, however. The code has some size restrictions. Here's the preamble and some code from main in ./ownme:

```
root@rd-linux:~# objdump --section=.text                      \
         --start-address=0x08048450 --stop-address=0x080484d4 \
         -d ./ownme

./ownme:     file format elf32-i386

Disassembly of section .text:

08048450 <main>:
8048450:       55                      push   %ebp
8048451:       89 e5                   mov    %esp,%ebp
8048453:       83 ec 08                sub    $0x8,%esp
8048456:       90                      nop
8048457:       90                      nop
8048458:       90                      nop
...
80484d0:       c9                      leave
80484d1:       c3                      ret
```

Granted, main isn't always the entry point, but it's easy to find out what is by the e_entry attribute of the elf header. Now, the reason I say main is a great place to use as a dump zone is because it holds code that will _never be accessed again_. This is the key. There are lots of other places you could use as a dumpzone. For instance, if the application contains a large helper banner, you

could put code over the help banner considering the banner wont be printed ever again once the program is executing. Use your imagination, you'll think of lots more. 'main' is the most generic method, since it's guaranteed in every application.

Well, now we know where we can safely put code to be executed, but how do we actually execute it?

* How could we possibly change the course of execution?

In order to change the course of execution in a process you need some working knowledge of ptrace and how the vm traverses an executable. Assuming you have both, read on. On x86 there is a vm register used to hold the vma of the NEXT instruction. Once an instruction finishes, the vm processes the instruction at eip (the vm register) and increments eip by the size of the current instruction. There are some instructions, such as jmp and call which are themselves execution diversion functions that cause eip to be changed to the address specified in the operand. We use this same principal when it comes to changing our course of execution to what we want.

Now, let's say that we theoretically put some of our own code at 0x08048450 (the address of main above) using the functionality from the ``Memory Management`` section. In order to have our code get executed (since it would normally never get executed) we use ptrace's PTRACE_SETREGS and PTRACE_GETREGS functionality. These two methods allow a third party process to obtain the registers and set the registers of another process. These registers include eip. In order to change the execution we perform the following steps:

1) call PTRACE_GETREGS to obtain the 'current' set of registers.

2) set eip in the returned set of registers to 0x08048450 (the address of our code).

3) call PTRACE_SETREGS with our modified structure.

4) continue the course of execution.

We've now successfully caused our code to be executed, but there's a problem. We injected a small chunk of code that we wanted to be run, but then we wanted the process to return to normal execution. That brings us to the next question.

```
* How do we restore execution once our code has finished?

Glad you asked, because this is the most important part.  In order
to restore execution we need a to modify our injected code just
a bit in order to make it easy for us to restore execution.  We
do this by adding an instruction near the end:

   int $0x3

This is on Linux (and Windows) to signal an exception or breakpoint
to the active debugger.  In the case of Linux, it sends a SIGTRAP,
which, if the process is being traced will be caught by wait().

Okay, so we've modified our code and let's say it looks something
like this:

   nop
   nop
   nop
   nop
   nop
   nop

   mov $0x1, %eax
   int $0x3
   nop

The code is setup with a 6 byte nop pad at the top to make our
changing of eip more cleaner (and safer) due to the way the vm
reacts to our execution diversion.  The movement of 1 into
eax is just an example of our arbitrary code.  The int $0x3
alerts our attached debugger (ptrace) and the nop is for padding
so we can see when we hit the end of our code.

Okay, that's a lot of stuff.  Let's walk through our modified
process of execution now.  This assumes you've already injected
your code at main (0x08048450):

   1) call PTRACE_GETREGS to obtain the 'current' set of
      registers

   2) save these registers in another structure.  This is used
      for restoration.

   3) set eip in the returned set of registers to
      0x08048450 (the address of our code).
```

```
        4) call PTRACE_SETREGS with the modified structure.

        5) continue execution, but watch for signals with the wait()
             function.  If the wait function returns a signal
           that is a stop signal:

           a) call PTRACE_GETREGS and get the current set of registers

           b) if eip is equal to the size of your injected code - 1
              (the location of the nop at the end), you know you've
              reached the end of your code.  go to step 6 at this
              point.

           c) otherwise, continue executing.

        6) at this point your code has finished.  call PTRACE_SETREGS
           with the saved structure from step 2 and you're finished.
           you've successfully diverted and reverted execution.

    That was a mouthful, but it's very important that it's understood.
    All of the topics in this document emplore this underlying
    logic to perform their actions.  Each one has a 'stub' assembly
    function that gets injected into a process at main to be executed.
    This code is meant to be small due to the fact that there are
    potential size issues.

    Oh, and another thing, you have full control over every register
    in this scenario because the registers are restored with
    PTRACE_SETREGS before the 'normal' execution continues.

[-- 2) Memory allocation                                          --]

    Memory allocation is one of the key features in this documented
    as all of the sub topics in Execution Diversion are dependant
    on its functionailty.  Memory allocation allows for dynamic
    memory allocation in another process (duh).  The most applicable
    scenario with regards to this document for such a thing are the
    storage of arbitrary code in memory without size limitations.
    This allows one to inject a very large function for execution
    without having fear that they will overrun into another function
    or harmful spot.

    Memory allocation is relatively simple, but understanding how to
    get from a to b requires a bit of explaining.  The first thing we
    need to do is figure out where malloc will be in a given process
    image so that we may call into it.  If we can figure that out
    we should be home free considering what we know from section 1.3.
```

```
Realize that all these steps below can and are easily automated,
but for sake of knowing, here they are:

   1) Where could malloc possibly be?  Well, let's see what
      our choices are:

      root@rd-linux:~# ldd ./ownme
         libc.so.6 => /lib/libc.so.6 (0x40016000)
         /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

      root@rd-linux:~# objdump --dynamic-syms --section=.text \
                       /lib/libc.so.6 | grep malloc
         0006df90  w   DF .text  00000235  GLIBC_2.0   malloc
      root@rd-linux:~# objdump --dynamic-syms --section=.text \
                       /lib/ld-linux.so.2 | grep malloc
         0000c8f0  w   DF .text  000000db  GLIBC_2.0   malloc

      Alright, so we've got malloc in both libc and ld-linux.  We
      could probably use either but what about programs that don't
      use libc?  In order to be the most flexible, we should use
      ld-linux.  This also has a positive side effect which is
      that every elf binary has an 'interpreter', and, it just
      so happens to ld-linux is that interpreter.

   2) Alright, so we know the vma of malloc is at 0x0000c8f0,
      but that doesn't exactly look like a valid vma.  That's
      because it's not.  It's an offset.  The actual vma
      can be calculated by adding the base address from ldd
      for ld-linux (0x40000000) to the offset (0x0000c8f0)
      which, in turn produces the full vma 0x4000c8f0.  Now
      we know exactly where malloc is.

   3) Cool, so we know where malloc is, now all we need to do is
      divert execution to some code that calls it and revert back.
      We also need the return address from malloc though so we
      know where our newly allocated buffer is at.  Fortunately,
      this is quite easy with PTRACE_GETREGS.  eax will hold the
      return value (cdecl).  The code is pretty simple and,
      considering we control all the registers, we can use
      them to pass arguments, such as size, into our code
      at the time of diversion.  Here's some code that will,
      when diverted to with the correctly initialized registers,
      call malloc and interrupt into the debugger:

         nop              # nop pads
         nop
```

```
            nop
            nop
            nop
            nop

            push %ebx        # push the size to allocate onto the stack
            call *%eax       # call malloc
            add  $0x4, %esp  # restore the stack
            int  $0x3        # breakpoint
            nop
```

The above code expects the 'size' parameter in ebx and the address of malloc in eax.

4) Alrighty, so now we've executed our code and we're ready to restore the process to normal execution, but wait, we need the address malloc returned. We simply use PTRACE_GETREGS and save eax and we've successfully allocated memory in another process, and we have the address to prove it.

The same steps above can be used for deallocating memory, simply s/malloc/free/g and you're set :).

[-- 3) Memory management --]

I'm only going to briefly cover the concept of copying memory from one process to another as it's sort of out of the scope of this document. If you're more curious, read about memgrep in the ``References`` section.

Copying memory from one process to another simply entails the use of PTRACE_POKEDATA which allows for writing 4 bytes of data to a given address inside a process. Not much more is needed to be known from that point on :).

[-- 4) Library Injection --]

Library injection is very powerful when it comes to using functionality inside a running process that it wasn't meant to be doing. One of the more obvious applications is that of loading a personally developed shared object into a running executable.

This one was fun to figure out, so I'll just kind of walk you through the process I took.

First thing's first, we need to figure out how to load a library

without the binary being linked to libdl. libdl is what provides
functions like dlopen(), dlsym(), and dlclose(). The problem is
that executables don't link to this library by default. That means
we can't do our magic technique of figuring out where dlopen will
be in memory because, well, it isn't guaranteed to be there.

There's still hope though. dl* functions are mainly just stubs
that make calling the underlying API easier. Kind of like how
libc makes calling syscalls easier. Since these are just wrappers,
there have to be implementers, and indeed, there are. Check this
out:

```
root@rd-linux:~# objdump --dynamic-syms /lib/libc.so.6 | \
                  grep _dl_ | egrep "open|close|sym"
000f7d10 g    DF .text  000001ad  GLIBC_2.2   _dl_vsym
000f6f10 g    DF .text  000006b8  GLIBC_2.0   _dl_close
000f6d80 g    DF .text  00000190  GLIBC_2.0   _dl_open
000f7c00 g    DF .text  0000010d  GLIBC_2.2   _dl_sym
```

Well, isn't it our lucky day? libc.so.6 has _dl_open, _dl_sym, and
_dl_close. These look amazingly similar to their dl* wrappers.
In fact, they're almost exactly the same. Compare the prototypes:

```
extern void *dlopen (const char *file, int mode);
extern void *dlsym (void *handle, const char *name)
extern int dlclose (void *handle);
```

To:

```
void *_dl_open (const char *file, int mode, const void *caller);
void *_dl_sym (void *handle, const char *name, void *who);
void _dl_close (void *_map);
```

Pretty much the same right? Looks very promising. So here's what
we know as of now:

* We know where the _dl_* symbols will be at in the processes
  virtual memory. (We can calculate it the same way we did
  malloc)
* We know the prototypes.

One thing we don't know is how the functions expect their arguments.
One would think they'd be stack based, right? Well, not so. They
seem to use a variation of fastcall (like syscalls). Here's a
short dump of _dl_open:

```
000f6d80 <.text+0xdde00> (_dl_open):
```

```
     f6d80:       55                      push   %ebp
     f6d81:       89 e5                   mov    %esp,%ebp
     f6d83:       83 ec 2c                sub    $0x2c,%esp
     f6d86:       57                      push   %edi
     f6d87:       56                      push   %esi
     f6d88:       53                      push   %ebx
     f6d89:       e8 00 00 00 00          call   0xf6d8e
     f6d8e:       5b                      pop    %ebx
     f6d8f:       81 c3 ba 10 02 00       add    $0x210ba,%ebx
     f6d95:       89 c7                   mov    %eax,%edi
     f6d97:       89 d6                   mov    %edx,%esi
     f6d99:       89 4d e4                mov    %ecx,0xffffffe4(%ebp)
     f6d9c:       f7 c6 03 00 00 00       test   $0x3,%esi
     f6da2:       75 1c                   jne    0xf6dc0
     f6da4:       83 c4 f4                add    $0xfffffff4,%esp
```

Looks pretty normal for the most part right? Well, up until 0xf6d95 at least. It's quite odd that it's referencing eax, edx, and ecx which have not been initialized in the context of _dl_open, and then using them and operating on them later in the function. Very strange to say the least. Unless, of course, the arguments are being passed in registers instead of via the stack. Let's look at the source code for _dl_open.

```
void *
internal_function
_dl_open (const char *file, int mode, const void *caller)
{
     struct dl_open_args args;
     const char *objname;
     const char *errstring;
     int errcode;

     if ((mode & RTLD_BINDING_MASK) == 0)
          /* One of the flags must be set.  */
          _dl_signal_error (EINVAL, file, NULL,
            N_("invalid mode for dlopen()"));

   ....

}
```

Okay, so we see roughly the first thing it does is do a bitwise and on the mode passed in to make sure it's valid. It does the and with 0x00000003 (RTLD_BINDING_MASK). Do we see any bitwise ands with 0x3 in the disasm? We sure do. At 0xf6d9c a bitwise and is performed between $0x3 and esi. So esi must be where our mode is

```
stored, right?  Yes.  Let's see where esi is set.  Looks like it
gets set at 0xf6d97 from edx.  Okay, so maybe edx originally
contained our mode.  Where does edx get set?  No where in _dl_open.
That means the mode must have been passed in a register, and not on
the stack.

If you do some more research, you determine that the arguments
are passed as such:

   eax = library name (ex: /lib/libc.so.6)
   ecx = caller (ex: ./ownme)
   edx = mode (ex: RTLD_NOW | 0x80000000)

Alright, so we know how arguments are passed AND we know the address
to call when we want to load a library.  From this point things
should be pretty obvious.

All one need do is allocate space for the library name and the
caller in the image using the ``Memory Allocation`` technique.
Then copy the library and image using the ``Memory Management``
technique.  Then, finally, execute the stub code that loads the
library.  That code would look something like this:

   nop                  # nop pads
   nop
   nop
   nop
   nop
   nop

   call *%edi           # call _dl_open
   int  $0x3            # breakpoint
   nop

This code expects the arguments to already be initialized in the
proper registers from what we determine above and it expects
_dl_open's vma to be in edi.

Welp, we've successfully injected a shared object into another
processes image.  What you do from here is up to the desired
outcome.  Calling _dl_sym and _dl_close uses the same code as above,
but their arguments are as follows:

_dl_sym expects:

   eax = library handle opened by _dl_open
   edx = symbol name (ex: 'pthread_create')
```

```
    _dl_close expects:

       eax = library handle opened by _dl_open
```

[-- 5) Code Injection --]

I must say we're getting rather hardcore, we can allocate memory, copy memory and load shared objects into arbitrary processes. What more could we possibly want? How about some arbitrary, controlled code execution that isn't limited by size? Sounds spiffy!

[-- 5.1) Forking --]

Let's say we want to fork a child process inside the context of another process and have it execute an arbitrary function that we've allocated and stored in the processes memory image via the ``Memory Allocation`` and ``Memory Management`` methods. Doing the fork is as simple as writing up some code that will use ``Execution Diversion`` to fork the child and return control to the parent as if nothing happened. An example of forking and executing a supplied function is as follows:

```
       nop                  # nop pads
       nop
       nop
       nop
       nop
       nop

       mov  $0x2, %eax      # fork syscall
       int  $0x80           # interrupt
       cmp  $0x00, %eax     # is the pid stored in eax 0? if so,
                            # we're the child
       jne  fork_finished   # since eax wasn't zero, it means we're the
                            # parent.  jmp to finished.
       push %ebx            # since we're the child, we push the start
                            # addr
       call *%edi           # then we call the function
       mov  $0x1, %eax      # exit the child process
       int  $0x80           # interrupt
    fork_finished:
       int  $0x3            # we're the parent, we breakpoint.
       nop
```

This code expects the following registers to be set:

```
      ebx = the argument to be passed to the function
      edi = the vma of the function call in the context of the child.

   Forking is really as simple as that.   Now, one side effect is that
   if the daemon does not expect fork children (ie, it doesn't call
   wait()) then your child process will show up as defunct when it
   exits due to not being cleaned up properly.  There are ways around
   this, though.  You could use the ``Execution Diversion`` technique
   to perform cleanup of exitted children after for the process.

[-- 5.2) Threading                                                --]

   Similar to forking, but different by the fact that a thread runs
   in the context of the caller and shares memory, threading allows
   for pretty much the same things that forking does.  There are
   some risks with threading though.  For instance, it is _NOT_ safe
   to create a thread in a process that does not natural thread.  This
   is for multiple reasons -- the most important being that the
   threading environment is setup at load time (in the case of
   pthreads).  If Linux didn't use some ghetto application-level
   threading architecture, things wouldn't be so bad.

   If you really do want to take the risk of creating a thread,
   the process would be something like this:

      1) Inject libpthread.so into the process (``Library Injection``)
      2) Find pthread_create's vma in the process
         (``Library Injection``)
      3) Allocate and copy user defined code (``Memory Allocation``)
      4) Perform ``Execution Diversion`` on the stub code to
         create the thread.  An example of such code is:

         nop                  # nop pads
         nop
         nop
         nop
         nop
         nop

         sub  $0x4, %esp      # space for the id
         mov  %esp, %ebp      # store esp in ebp for pushing
         push %ebx            # push argument
         push %eax            # push function
         push $0x0            # no attributes
         push %ebp            # push addr to store thread id in
         call *%edi           # call pthread_create
```

```
      add  $0x14, %esp     # restore stack
      int  $0x3            # breakpoint
      nop
```

Like I said, threading is dangerous. Know your program before attempting to inject a thread. You will get odd results if you inject a thread into a process that doesn't naturally thread.

[-- 5.3) Function Trampolines --]

Function trampolines are a great way to transparently hook arbitrary functions in memory. I'll give a brief overview of what a function trampoline is and how it works.

The basic jist to how function trampolines work is that they overwrite the first x instructions where the size of the x instructions is at least six bytes. The six bytes come from the fact that on x86 unconditional jumps take up 6 bytes in opcodes. The x instructions are replaced with the jmp instruction that jumps to an address in memory that contains the injected function. This function runs before the actual function runs, and thus, has complete control over whether the actual function even gets called. At the end of the injected function the x instructions are appended as well as a jump back to the original function plus the size of the x instructions. Here's an example:

Let's say we want to hook the function 'testFunction' in the executable 'ownme'.

```
   root@rd-linux:~# objdump -d ownme --start-addr=0x080484d4

   ownme:     file format elf32-i386

   Disassembly of section .init:
   Disassembly of section .plt:
   Disassembly of section .text:

   080484d4 <testFunction>:
     80484d4:       55                      push   %ebp
     80484d5:       89 e5                   mov    %esp,%ebp
     80484d7:       83 ec 18                sub    $0x18,%esp
    ...
     8048500:       c9                      leave
     8048501:       c3                      ret
```

Well, it looks like the first 3 instructions match our criteria of at least 6 bytes. Let's keep those 6 bytes of opcodes

```
tucked away for now.

We need to be smart here.  We're going to do a jmp that
says jmp to address stored in address x.  We're also
going to want to restore back to the original place.  That means
when we allocate our memory we should allocate it in a format like
this:

   [ 4 bytes storing the address of our code                ]
   [ 4 bytes storing the address to jmp back to             ]
   [ X bytes of arbitrary code                              ]
   [ X bytes containing the X instructions that we overwrote ]
   [ 6 bytes for the jump back                              ]

So let's say we want to inject this code and we allocated
a buffer in the process of the approriate length which starts
at 0x41414140:

   nop
   movb $0x1, %al

Our actual buffer in memory would look something like this

   0x41414140 = 0x41414148 (address of our code)
   0x41414144 = 0x080484d8 (address to jmp back to)
   0x41414148 = 3 bytes (nop, movb)
     0x4141414B = 6 bytes of preamble from testFunction
     0x41414152 = jmp *0x41414144

The last step now that we have our code injected is to overwrite
the actual preamble (the 6 bytes of testFunction) with the jmp
to our code.  The assembly would look something like this:

   jmp *0x41414140  # Jump to the address stored in 0x41414140

Once that's overwritten, we're home free.  The flow of
execution goes like this:

   1) Call to testFunction
   2) First instruction of testFunction is:
      jmp *0x41414140
   3) vm jumps to 0x41414148 an executes:
      nop
      movb $0x1, %al
      push %ebp
      mov  %esp, %ebp
      sub  $0x18, %esp
```

```
            jmp  *0x41414144
         4) vm jumps to 0x080484d8
         5) Function executes like normal.

      That's all there is to it.  There are a couple of restrictions
      when using trampolines:

         1) NEVER modify the stack without restoring it before
            the original functions preamble gets called.  Bad
            things will happen.
         2) Becareful what registers you modify.  Some functions
            may use fastcall.

      For more information on function trampolines, see the ``References``
      section.

[-- 6) Conclusion                                                --]

      That about wraps it up.  You now have the tools to allocate,
      copy, inject libraries, create forks, create threads, and
      install function trampolines.  You also have the underlying
      concept of ``Execution Diversion`` which can be applied across
      the board to even more things I haven't even thought of yet.

[-- 7) References                                                --]

      * For information about ``Function Trampolines``:

         http://research.microsoft.com/sn/detours
```
