Sandboxing of Dynamic Code

Rune Hammersland
Student at NISlab

Sandboxing of code is convenient in a lot of situations, not only for evaluating code without the risk of contaminating the surrounding code. This document tries to get an overview of how this can be achieved in the dynamic languages Ruby, Perl and Python. It also contains a brief section on how some operating systems makes it easier running untrusted code using separation techniques resembling virtual machines.

Categories and Subject Descriptors: D.1.m [**Programming Techniques**]: Miscellaneous; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.6 [**Software Engineering**]: Programming Environment; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Languages, Security

Additional Key Words and Phrases: Dynamic languages, Ruby, Perl, Python, Sandboxing, Security

This document will discuss Sandboxing of code, and focus on the possibilities in the dynamic languages Ruby, Python and Perl. The reasons for the choice of languages are as follows:

(1) I am currently using Ruby as my primary programming language, and have been following discussion on the Sandbox module being developed by "why the lucky stiff" and "MenTaLguY"[1] (along with others).
(2) Dynamic languages are a big target for sandboxing as evaluating foreign code in these languages are both easy and dangerous.
(3) All these languages includes mechanisms to deal with running untrusted code. They are also very similar, as the creators of both Ruby and Python have looked to Perl for inspiration. Even though my knowledge of the others are restricted, it won't be too hard to read up on it.
(4) The languages are accessible for many people. All three are for instance included in a basic install of Mac OS X (and Perl and Python are available on most standard Linux installs). In addition all are available as free software [Free Software Foundation 2006], and their licenses are compatible with the GNU GPL [Free Software Foundation 1991].

1. WHAT IS SANDBOXING?

In this document Sandboxing will refer to a technique which allows you to execute untrusted code in your environment, without letting the untrusted code tamper with your surrounding code. We want to make sure the untrusted code cannot alter the way the surrounding program behaves. Ruby for instance has the principle of Open Classes, which means that it is possible to redefine all classes and their methods after they have been defined. Of course tampering with the data structure of the

[1]It is nearly impossible to find their real names, and god knows I've tried ...

surrounding code is also possible through the object's getters and setters. By using the sandbox we can separate trusted and untrusted code in such a way that the trusted code can tamper with the untrusted code, but not the other way around.

It is also a plus if the sandbox prohibits the untrusted code from wreaking havoc on your filesystem by i.e. restricting IO calls to a temporary filesystem, or prohibiting IO calls all together. We will almost always want to restrict access to the filesystem, as it makes it possible to tamper with important files (write something bogus to them, change the content with your own to make more illegal code run, or deleting it) and reading sensitive information (as the password file for instance).

A sandbox will rarely prohibit the untrusted code from creating an infinite loop which will steal all the CPU cycles, and in effect halting the main program. In effect working as a Denial of Service attack. Sometimes it is possible to work around this by running the sandbox in a thread which we kill after a specified time interval. This is not looked heavily into in this paper.

Virtual machines can also be used as sandboxes when you are running code you don't neccecarily trust. However, this is pretty cumbersome on today's systems, as it requires setting up and booting a virtual machine first. In the future this will probably be a lot simpler, especially due to the fact that dual (and quad) processors are becoming more and more common. These systems might for instance supply system calls to create a VM on the fly, and destroy this after you are finished using it.

The Java programming language uses a sandboxing approach to deal with running of possibly unsafe applets on the host machine. Java applets are a nice way of sharing programs over the internet, as people only need to navigate to a website and run the program from there. If the programmer updates the program, the users will always be running the latest version. The problem arises when you are running applets from sources you don't know, and hence cannot trust full access to your computer. To solve this, the Java VM (Virtual Machine) runs the applets in a sandbox, which prohibits them from accessing sensitive parts of the system (i.e. they only get to write and read files from a temporary directory created for the applet).

2. WHY WOULD YOU WANT TO SANDBOX CODE?

There can be several reasons to why you would like to sandbox a piece of code. Often it is because the code is untrusted, but there are other uses as well. I'll present a couple here:

—Run tests against two different versions of a library (discussed on page 10)

—Running two instances of a program in the same parser without worrying about namespace clashes, and the possibility of an instance tampering with the other instance.

—Copy the environment a program runs in (which also opens up the possibility of serializing the whole environment).

Take for instance the popular web framework Ruby on Rails. This framework allows for rapid development and quick prototyping, but has the limitation that an application has to run in an instance of rails (which itself is a Ruby program). Using sandboxing it is possible to launch one Ruby runtime, which holds a number of sandboxes, each with it's own rails instance. This enables you to run several rails webapps in the same Ruby runtime without having to worry about one redefining behavior in the other.

If the sandbox enables you to copy and serialize (store the contents somewhere for later reading) the current environment, you could easily add the possibility for "snapshots" in your application where it is possible to restore the complete state of the program. To use a silly example (silly because this is a silly way to implement it), you could write a game where the user is allowed to save his or her progress at any time. When saving you could simply serialize the current state of the sandbox the game is running in, and when the player loads a game later, you could load the serialized sandbox. This could be used when prototyping the game, before you actually wrote the save / load functionality.

Using the same approach you could also add backstepping abilities to a debugger by serializing the environment for every n steps. If the user now wants to step back in the execution, this is possible by restoring the previously serialized state of the sandbox. There are more efficient ways to do this, but by using the sandbox approach you will be up and running in a short period of time. Back stepping is a feature few debuggers have.

3. SANDBOXING IN THE PROGRAMMING LANGUAGES

In this section, I'll be looking into ways of achieving sandboxing through mechanisms of the programming languages covered in this paper. All three has a way of achieving sandboxing through the standard functionality in the language (without the use of extra modules), and in Ruby's case, work is being done on a new module which eventually will enter the standard Ruby distribution [Matsumoto 2006a].

3.1 Python

In Python there are basically two ways of evaluating code: `eval` and `exec`. While `eval` evaluates an expression in a string or a code object and returns the value of the statement, while `exec` executes a statement. For non native English speakers this needs further explanation:

`eval` only evaluates expressions (i.e. "1 + 2"), and cannot execute statements (program code). While `exec` of course can evaluate an expression, it has no use, since nothing is returned from the `exec` function, and the only way of knowing the result of that expression is to assign it to a variable in the statement itself. Evaluating other people's code through `eval` is therefore "safe". Evaluating other people's code through `exec` is not, as it can modify your own variables, and do all kinds of malicious activities. This is different from how things are done in i.e. Ruby, where only `eval` exists, but it does the same thing as both `eval` and `exec` in Python.

In Python you can also call `exec` within a `Dictionary` (similar as a `Hash` in other languages) instead of the global namespace. This limits the variables accessible to the variables in the `Dictionary`. This, however, does not restrict access to malicious methods which for instance modifies files on the local filesystem.

3.1.1 *Restricted Execution.* Python has support for sandboxing in it's standard library. The pythonistas prefer to refer to it as Restricted Execution[Python Software Foundation], and not Safe Python or anything like that. This is because they acknowledge that *safe* can mean a lot of things, and that making sure something is *safe* is very hard to guarantee.

The book Python in a Nutshell[Martelli 2003] covers Restricted Execution pretty good, and is a worthwhile read. Using `RExec` you can create a sandbox for the untrusted code to be run in. In this sandbox you can add and remove modules, and also replace existing modules with your own "safe" substitutions (i.e. a `open()`-method that cannot open files for writing or appending, or only opens files in a specific directory).

Since code executed or evaluated in the `ERxec`-environment runs in a sandbox, it cannot modify the code which is already running in the interpreter. It can, however, run code that creates exceptions. Also, the sandbox raises exceptions if the imported code tries to do something it's not allowed to. This means that if you have a running program which tries to execute untrusted code in a sandbox, you should probably do this in a `try/except`-block. If you don't do this, raised exceptions will terminate execution of your running program (and could of course be used as a Denial of Service-attack).

Let's take a look at how we can create a sandbox and run code in it. We will also add a module to the sandbox, so functionality from that module will be available:

Listing 1. Creating a sandbox in Python

```python
import rexec
try:
    sandbox = rexec.RExec()
    sandbox.add_module(math)
    sandbox.r_exec("print 'LOL from the sandbox!'")
    sandbox.r_exec(variable_containing_code_of_some_sorts)
except:
    print 'Caught exception ...'
```

We could also unload modules using `sandbox.r_unload()`. If you want to have more control over import statements, you can supply an argument to the class when you are instantiating it. Giving a method there, using the `lambda` method, the module will be imported only if the method you supplied returns `True`. If you want to restrict the sandbox further, you can do this by changing some attributes on the sandbox. These attributes are `tuples` (arrays) of strings containing names of functions, modules and directories to be allowed or denied in the sandbox. The properties are as follows:

nok_builtin_names. Built-in functions which should not exist in the sandbox.

ok_builtin_modules. Built-in modules that are okay to import in the sandbox.

ok_path. The loadpath for the sandbox.

ok_posix_names. Attributes of the operating system that are okay to use in the sandbox.

ok_sysnames. Attributes of `sys` that are okay to import in the sandbox.

3.1.2 *Removing "unsafe" methods.* Python also has a module called `Bastion`, which can wrap objects and forward method calls to only those methods you assume to be “safe”. This is done by supplying a filter to the bastion-object which returns true for the methods that should be executable by the untrusted code. Using some additional code, you can create a factory method which returns “safe” objects of the class to be safeguarded.

Listing 2. Putting a class in a Bastion

```python
import rexec, Bastion

# Example class to include in the sandbox.
class Example:
    def __init__(self, str):
        self.str = str

    # This method only prints a variable, so we concider it safe.
    def safe_method(self, arg):
        print self.str + arg

    # This method modifies the variable, so we concider it unsafe.
    def unsafe_method(self, arg)
        self.str = self.str + arg

# Filter to differentiate between the good and the bad.
def safe_filter(method):
    if method[0] == "_" or method[0:6] == "unsafe": return False
    else: return True

def class_factory(arg):
    return Bastion.Bastion(Example(arg), safe_filter)

# Create sandbox, and include the ``bastioned'' version of our class.
sandbox = rexec.RExec()
sb_builtins = sandbox.add_module('__builtins__')
sb_builtins.Example = class_factory
```

3.1.3 *A note on Python 2.3 and 2.4.* Python 2.3 and 2.4 raises an exception when you try to instantiate a `RExec` or `Bastion` class. This is because these modules have been deprecated in Python 2.3 [Python Software Foundation 1998]. This is because the new way classes are designed in Python $\geq$ 2.3 offers several ways to break out of the restricted execution environment. The Python developers have stated that they don't have the time or the interest to fix this for the time being. They also note that earlier versions of Python have known bugs in the `RExec` module, so other ways to achieve this should be considered.

3.2 Perl

Perl lets you evaluate statements using the `eval` and `do` routines. The only difference between these is that `eval` evaluates the contents of a string, while `do` evaluates the contents of a file. Sometimes this is something we actually want to do, and as explained before, using these routines on strings and files you don't know the contents of should not be done without taking precautions.

Perl offers a couple of ways to help you in running other people's code (and of course other security mechanisms as well). While not being used for running (or securing) code, Perl has a technique called tainting. With tainting, all strings that are user supplied (either via reading of stdin, or as arguments to the program, etc.) are tainted. Tainted strings cannot be used for dangerous routines (i.e. you cannot open a file if the string containing the path is tainted). Tainting is turned on automatically if the program is run `setuid` or `setgid`, and can be turned on by using the `-T` argument to Perl. Unfortunately checking for taintedness in Perl is not straight forward (but then again, you should assume objects being tainted if running in tainted mode and calling unsafe routines). Listing 3 shows two methods, the first taken from Programming Perl [Wall et al. 1996], and the second from CGI Programming with Perl [Guelich et al. 2000]. It also shows how untainting is done [Wall et al. 1996].

Listing 3. Checking for taintedness

```
# kill tests for taintedness even when no PID to signal is supplied.
sub is_tainted {
    not eval {
        join ("", @_), kill 0;
        1;
    };
}

# Use a substring containing nothing, and use eval for taint checking.
sub is_tainted {
    my $var = shift;
    my $blank = substr ($var, 0, 0);
    return not eval { eval "1 || $blank" || 1 };
}

# Untainting a value:
# (Perl assumes matches from regular expressions to be safe)
if ($addr =~ /^([-\@\w.]+)$/) {
    $addr = $1;
} else {
    die "Address is unsafe."
}
```

3.2.1 *Safe.pm.* `Safe.pm` is a Perl module which creates a "container", or what we will refer to as a sandbox, for code to run in. At creation, this sandbox will get it's own namespace, so it will not have access to the surrounding code. The only variables that are shared are the variables `%_`, `$_` and `@_`. This is because a lot of standard functionality works on these variables and most of Perl would break if they were not available.

After creating a sandbox (or even when creating it), you can specify a operator mask for the sandbox. This is basically a string containing `0x00` or `0x01` values, and has as many characters as there are operators in the Perl environment [2]. If a `0x01` is found, the offset it was at denotes the operator number to mask off (or deny access to if you like). There are a couple of convenience routines that make creating and maintaining this string a bit easier. After creating a sandbox, you could i.e. call `$sandbox->trap("read");` or `$sandbox->untrap("read");` to trap or untrap this operator. There is also a `ops_to_mask` routine which takes a list of operators, and creates a mask where these are masked out. The default mask will mask out all operations that in some way gives access to the system, except a few like `sysread` which needs a filehandle to operate anyways (which you can supply to the sandbox if needed).

To evaluate code in the sandbox, you can use the `Safe` object's `reval` and `rdo` routines (shown briefly in listing 4). These routines work exactly like you expect them to: evaluating a string or the contents of a file in the sandbox. If something fishy happens an error will occur. This error will occur run-time for the surrounding code, but compile-time for the sandboxed code (unless the sandboxed code contains an `eval` statement, in which case it will happen run-time in the sandbox as well, but compile-time for the nested `eval`). If an error occurs, the `$@` vector will contain an error message along the lines of ``` ``%s trapped by operation mask operation ...'' ```, and the code will not be run. If no error occurs, the `reval` or `rdo` routine will return the last statement executed, or the returned value.

It is also possible for the surrounding code to share variables (and subroutines as these can be accessed by a variable) with the sandbox. This is done using the sandbox's `share` routine, as seen in listing 4. The routine takes a string containing the name of the variable (with a leading type identifier) to share. In this way you can create semi-safe subroutines for exceptions where you want code in the sandbox to be able to do something that will usually be restricted in the sandbox. This can be done since the operator mask in the sandbox only applies to code being compiled, and as the subroutine you're sharing has been compiled in the surrounding code (where there is no, or a different, operator mask) calling it will work perfectly fine.

Listing 4. Sharing code with the sandbox

```perl
my $sandbox = new Safe;
my $untrusted_file = "sample.pl"

sub potentially_unsafe {
    @_; # Untaint arguments found in the argument vector.
    system ('rm', 'foo', 'bar'); # Call unsafe methods.
}

# Share the method with the code in the sandbox.
$sandbox->share ('&potentially_unsafe');
$sandbox->reval ('print $_;');
$sandbox->rdo ($untrusted_file);
```

[2]Note that operators here mean core operations, and not operators as we know them from i.e. mathematics

3.3 Ruby

In Ruby it is possible to evaluate expressions and statements using the built-in method `Kernel.eval` (usually you can drop the `Kernel` part, as methods not found in other objects will be looked up in `Kernel`). Ruby's `eval` method is very powerful and hence extremely dangerous to use if you are not 100% sure about what you are doing. For instance it is no problem to "require" code found on a website in your running application using a block of code like this:

Listing 5. Requiring a library from the Internet

```
# open-uri allows us to open() websites (among other things)...
require 'open-uri'

begin
    require 'foo-lib'
rescue LoadError
    puts "foo-lib not found on your system.",
         "Getting it from the internets ..."
    eval open('http://foo-lib.org/foo-lib/current/foo-lib.rb').read
end
```

While this is perfectly possible to do, it of course opens up a whole bunch of possible security holes. Even if you trust the developer of `foo-lib`, and feel totally safe that the path always will point to the current version of the library, you could always be a victim of a DNS poisoning attack, and without knowing it require another library (or program statements). To use a famous phrase: *With great power, comes great responsibility* [Lee et al. 1963].

While Ruby doesn't have a sandbox module in it's core libraries yet (stable version as this is being written is version 1.8.5), it does offer a way to make your programs more safe. A sandbox is under development, and Ruby's author, Yukihiro Matsumoto, has been very helpful, allowing the developers access to Ruby's internals [Matsumoto 2006b] to make the development possible [Matsumoto 2006c] [3]. He has announced that he would like to have it in the standard distribution of Ruby, once it becomes stable [Matsumoto 2006a].

3.3.1 *$SAFE.* Ruby, as BSD, has the concept of safe levels (called securelevels in BSD). Safe levels can be adjusted either in code, or while invoking the interpreter. If the interpreter is started with `ruby -T3 scriptname` or `ruby -T 3 scriptname`, the safe level is set to 3. In addition: if the script is run `setuid` or `setguid`, the safe level is automatically set to 1. Setting the safe level in the source code is as easy as `$SAFE = 2`. Note that you cannot lower the safe level. Trying to do so will yield a SecurityError exception.

A table explaining what the different safe levels do is included in the Pickaxe [Thomas et al. 2004] (a longer version also exists in the same book), and included here:

[3]Note that this is not referring to making the source code available, as Ruby is published under a MIT like lisence. It means making variables and data structures available to other C libraries. Thus making it possible to use the sandbox module without patching Ruby itself

$SAFE	**Constraints**
0	No checking of the use of externally supplied (tainted) data is performed. This is Ruby's default mode.
≥ 1	Ruby disallows the use of tainted data by potentially dangerous operations.
≥ 2	Ruby prohibits the loading of program files from globally writable locations.
≥ 3	All newly created objects are considered tainted.
≥ 4	Ruby effectively partitions the running program in two. Nontainted objects may not be modified. Typically, this will be used to create a sandbox: the program sets up an environment using a lower $SAFE level, then resets $SAFE to 4 to prevent subsequent changes to that environment.

Table I. Constraints in different safe levels

An explanation of tainting is found in the Perl section (section 3.2). Tainting in Ruby is similar to tainting in Perl, but in Ruby checking and untainting is a bit easier. To check if an object is tainted, you only have to call the `tainted?` method, and to untaint the object, all you have to do is call the `untaint` method (after checking if it is safe of course). Duplicating a tainted object, or concatenating it with another (as you're likely to do with strings) will yield another tainted object. As you can see, using safe level 4, it is impossible to modify objects that are untainted (in effect locking them down). This can be, and has been, used to create sandboxes.

Listing 6. Demonstrating $SAFE

```
def try_insecure level=1, &blk
    # Supersimple "sandbox"
    Thread.new {
        $SAFE = level
        begin
            yield blk
        rescue SecurityError => e
            puts "Caught SecurityError: #{e}"
        end
    }.join
end

# input is tainted, so supplying it to IO#open yeilds Security Error.
try_insecure 1 do
    puts "Write something: "
    input = gets.chomp # Tainted input

    File.open input do |file| # raises exception here.
        puts f
    end
end
# Using SAFE level 3, newly created objects are considered tainted.
# In effect: opening files with new strings will not work. We will need
# a string created before raising the $SAFE level ...
try_insecure 3 do
    File.open 'BOOYA', 'w' do |file| # raises exception here.
        file.puts "hehe"
    end
end
```

3.3.2 *Sandbox.* This summer, work started on a sandbox for Ruby, written in "Ruby C". There have been earlier attempts at making sandboxes (usually using safe levels), one is even mentioned in the Pickaxe [Thomas et al. 2004], but none of them has been written as a module in C. The FreakyFreakySandbox (as it was first referred to in the author's blog) has, as earlier mentioned, got Yukihiro Matsumoto's blessing, and will likely be included in the standard distribution of Ruby when it's done.

It works by creating a struct called `sandkit`, which holds a whole Ruby environment. In this environment some of the standard methods have been modified to make sure people cannot do evil things like calling `Kernel#fork`, `Kernel#system` and other "dangerous methods" (these methods are simply not defined). There are two kinds of sandboxes: Full and Safe. The safe one is more restricted (obviously), and both can take an argument `:timeout` to specify how long the code should be allowed to run before the sandbox shuts down.

An announcement posted to the ruby-core mailinglist [why the lucky stiff 2006] even includes code to show how you can load two versions of the same gem (a library packaged in a specified way) to do testing on multiple versions:

Listing 7. Loading different versions of a library

```
require 'sandbox'

hpricot = Sandbox.new :init => [:all]
hpricot.eval(%{
  require 'rubygems'
  require_gem 'hpricot', '=0.4.47'
  require 'hpricot'
  puts Hpricot("<a class=test>link</a>").search(".test")
})

hpricot2 = Sandbox.new :init => [:all]
hpricot2.eval(%{
  require 'rubygems'
  require_gem 'hpricot', '=0.4'
  require 'hpricot'
  puts Hpricot("<a class=test>link</a>").search(".test")
})
```

3.3.3 *Removing "unsafe" methods.* Removing access to "unsafe" methods in Ruby, can be done through `Module.remove_method` and `Module.undef_method`. The difference between these are that `Module.remove_method` removes the method from the class, but must be called in the class the method is implemented, and has no effect in derived classes (unless they reimplement the method, in which case the reimplemented method will be removed, and the inherited method will be called if something tries to call the method). `Module.undef_method` on the other hand only prevents the class (and derived classes) from responding to the method (the method is not actually removed).

Listing 8. Removing unsafe methods from a class in the sandbox

```
1  class Example
2      def safe_method(str)
3          puts @var.to_s + " " + str.to_s
4      end
5
6      def unsafe_method(str)
7          @var += str
8      end
9  end
10
11 sb = Sandbox.new; sb.import Example
12 sb.eval "Example.instance_eval { remove_method :unsafe_method }"
13
14 Example.instance_eval "{ remove_method :unsafe_method }"
15 sb = Sandbox.new; sb.import Example
16
17 sb = Sandbox.new; sb.ref Example
```

As you can see in listing 8, there are a couple of ways to achieve this. The first example (line 11 and 12) will remove the method only in the sandbox. The second approach (line 14 and 15), will remove the method from the surrounding code as well, but will result in sandboxed code being unable to call it as well. The problem with removing methods from classes in the sandbox, is that it's not really fool proof. A malicious user could always reimplement the functionality in the sandbox (unless it relies on C extensions or other required files which may not be included in the sandbox).

There is also the third approach (line 17). The difference between `Sandbox#import` and `Sandbox#ref` is that `Sandbox#import` will copy the supplied class into the sandbox using a deep copy (copying parent classes/modules until it reaches `Object` or a class/module already defined in the sandbox). `Sandbox#ref` will create a reference to a class (or module) outside of the sandbox. This is implemented [why the lucky stiff 2006] by creating a `BoxedClass` in the sandbox which catches method calls through the `method_missing` method. When a method is called on a `BoxedClass`, the sandbox halts and "proxies" the method call to the surrounding environment, calls it there, marshals the result, restores the sandbox, and return the unmarshaled result from the `method_missing` method. Thus removing methods on the `BoxedClass` will not have any effect (it might work using `undef_method`) as the call is proxied out of the sandbox anyways. `Sandbox#ref` is meant for supplying things like restricted database access (see the reference for more details).

Unfortunately, I have not had the chance to build the sandbox module on my system to play around with code examples. This is mainly because I wrote the Ruby section last, but also because to build it against Ruby 1.8.5 (which is the current version), you need to patch Ruby (thus requiring me to recompile Ruby itself as well), or build Ruby from CVS. In any case, I would be short on time.

4. SANDBOXING OUTSIDE OF THE PROGRAMMING LANGUAGES

Using security features of the operating system it is possible to create something resembling small virtual machines within the system. All UNIX like systems has this possibility using the `chroot` mechanism, which we will look into in the next chapter. FreeBSD (and some other BSD variants) also include a strengthening of the standard `chroot` mechanism. Unfortunately, these mechanisms are not supported "out-of-the-box" in Microsoft Windows, but there are some third party alternatives which I will quickly introduce.

4.1 UNIX: chroot

Using chroot, you are able to create a confined space in the filesystem in which untrusted programs may run without being able to access the sensitive parts of the filesystem. While this may sound good, there are several known ways to break out of a chroot environment (including making your own disk device and mount that), so you should not trust this approach fully, but rather think of it as an extra bump in the road. A simple way to break out is included (note that if there is no root user defined within the chroot environment, no `setuid` binaries, no devices and the process running in the environment dropped root privileges after entering the chroot, breaking out seems impossible):

Listing 9. Breaking out of chroot

```c
int main(void)
{
    int i;
    mkdir ("breakout", 0700);
    chroot ("breakout");
    // The new root is now one step further down in the file system, and
    // the program is now outside of the chroot (since the root is moved).
    // Change directory upwards until we reach the real root.
    for (i = 0; i < 100; i++)
        chdir ("..") ;
    chroot (".");
    // Give us a shell!
    execl ("/bin/sh", "/bin/sh",NULL);
}
```

`chroot` can be invoked through the `chroot` command, or through the `chroot` function in the C programming language. By using the command line tool, you can chroot programs that doesn't do this themselves using the C API. Through the command line tool you can also specify which user and group the program invoked in the chroot environment should run as. If you are using the C API, the same can be achieved using the functions `setgid()`, `setgroups()` and `setuid()`.

4.2 BSD: jail

Availiable in FreeBSD and in NetBSD. It is not included in OpenBSD, as Theo de Raadt claims it's too complicated to be secure. `jail` [Kamp and Watson 2000] works in a similar way to `chroot` (and actually uses `chroot` itself), but it includes several improvements with regard to security. It most notably restricts certain systemcalls, so it is impossible to mount anything, you cannot change network settings and more. To support all this, the jail system call (and hence the command line tool) gets some help from the system kernel. Opposed to `chroot`, breaking out of a `jail` has not been done, even with root privileges. Further, each jail created may only bind to one IP address, and some of the functionality in the network layer has been restricted (i.e. spoofing IP addresses). More information about what is restricted, and how things are implemented (including how the kernel and standard API was changed) can be found in the paper about jail [Kamp and Watson 2000].

BSD also includes a concept called securelevels (mentioned in section 3.3.1). Using these, you can lock down the system to different levels. An often used analogy is that to the Defcon-levels in the US armed forces. By raising the securelevel, you harden the system against attacks. Superusers can raise the securelevel, but only the `init` process (and not even that on FreeBSD) can lower it. When used in combination with `jails`, each jail gets it's own securelevel. The highest of the jail's securelevel and the global securelevel (that of the host system) will be used in the jail.

4.3 Solaris: Containers

Sun Solaris 10 contains a mechanism called "Containers" and "Zones". Using these, we are able to create virtual machines within the operating system. These virtual machines cannot interfere with the host (they have no access to the filesystem), and restrictions on CPU use and other hardware resources can be enforced per virtual machine. It is also possible to create "Sparse Zones", which includes the minimum it needs to run (to use less disk space), or "Whole-Root Zones" in which the whole operating system is duplicated. A "Container" is a collection of zones which shares the same hardware resources. Not everything is suited to run in "Zones", as they are not really running their own kernel. More info about "Zones" can be found in the man pages for `zoneadm` and `zonecfg`, and through Sun BluePrints [et. al. 2006].

4.4 Windows

There are no built in functionality in Microsoft Windows to achieve these things (it is in Windows Virtual Server, and will probably be in Windows Vista). There are however third party developers who develop solutions resembling BSD's `jail`. Trustware [4] has developed something called *BufferZone*, which hooks into the Windows kernel and intercepts "dangerous" systemcalls, and reroutes things like writing to the registry. Using this technique, it is possible to run unsafe programs in a `BufferZone` (or a sandbox if you will) to avoid problems affecting the host system. It also stores "unsafe writes" in another location, so everything looks consistent for the program being sandboxed.

In addition, it's worth mentioning Norman's attempt to fight computer virii using sandboxing [5]. Using this approach they can let untrusted programs run free in a sandbox, and find out if it is malicious depending on it's behavior.

[4] http://trustware.com
[5] http://sandbox.norman.no

5. CONCLUSION

The three languages I've looked at takes somewhat different approaches to sandboxing. The reason for this is the way the languages are built. In Perl for instance, everything gets compiled to a basic set of "operators" which you can then choose to disable. This might make it hard to know what routines will break, but you will be certain that the operation you prohibited will not be run from within the sandbox. While the approaches are different, there are some similarities. In all cases the main namespace is swapped with a new (more restricted). This is of course done to separate the sandbox and the surrounding code.

I found that all the three languages have (looking at the books at least) good support for creating separated execution environments. Sadly enough this is deprecated in Python. The only way to achieve similar things there is to `eval` the code in a restricted `dictionary`. Ruby's sandboxing module may not yet be complete, but it sure looks promising, and using the `$SAFE` variable (which has been available a long time) you can come a long way. Again; getting this right (and not firing off unnecessary many exceptions) can take some fiddling, but the possibility is certainly there.

Writing this paper has taught me more about the languages in question, especially Python and Perl. I have, as earlier stated, been using Ruby for a while now, but it was interesting to take a look at the "competitors". Noticing similarities between them, and also how being built different called for different solutions. Both Ruby and Python has elements inspired by Perl, so they are relatively similar syntax wise.

I have also learned not to trust books blindly, as the Python book [Martelli 2003] contained a great deal about the `Rexec` module, even though it was deprecated in the very same version of Python that the book covered. In addition, the Perl book [Wall et al. 1996] had a chapter about `Penguin`, a module for signing Perl code. This module has been "dead" since 1997. Not really the book's fault since it is from 1996, and having checked things twice, I didn't actually write anything about it.

A. SOURCES

REFERENCES

ET. AL., H. J. F. 2006. *The Sun BluePrints Guide to Solaris Containers: Virtualization in the Solaris Operating System*. http://www.sun.com/blueprints/1006/820-0001.html.

FREE SOFTWARE FOUNDATION. 1991. Gnu general public licence. http://www.gnu.org/copyleft/gpl.html.

FREE SOFTWARE FOUNDATION. 2006. The free software definition. http://www.gnu.org/philosophy/free-sw.html.

GUELICH, S., GUNDAVARAM, S., AND BIRZNIEKS, G. 2000. *CGI Programming with Perl*, Second ed. O'Reilly.

KAMP, P.-H. AND WATSON, R. N. M. 2000. Jails: Confining the omnipotent root. http://docs.freebsd.org/44doc/papers/jail/jail.html.

LEE, S., DITKO, S., AND KIRBY, J. 1963. *The Amazing Spider-Man 1*. Marvel Comics.

MARTELLI, A. 2003. *Python in a Nutshell*. O'Reilly.

MATSUMOTO, Y. 2006a. Re: [ann] sandbox 0.0.11 – taking the i out of eval. http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/202864.

MATSUMOTO, Y. 2006b. Re: [yay] my sandboxing extension!! http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/8314.

MATSUMOTO, Y. 2006c. Re: [yay] my sandboxing extension!! http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/8311.

PYTHON SOFTWARE FOUNDATION. *RExec Module in Python*. Python Software Foundation. http://docs.python.org/lib/restricted.html.

PYTHON SOFTWARE FOUNDATION. 1998. 17 new, improved and deprecated modules. http://www.python.org/doc/2.3.5/whatsnew/node18.html.

THOMAS, D., FOWLER, C., AND HUNT, A. 2004. *Programming Ruby: The Pragmatic Programmer's Guide*, Second ed. Pragmatic Bookshelf.

WALL, L., CHRISTIANSEN, T., AND SCHWARTZ, R. L. 1996. *Programming Perl*, Second ed. O'Reilly.

WHY THE LUCKY STIFF. 2006. Freakyfreaky now resumes its usual sandly self. http://redhanded.hobix.com/inspect/freakyfreakyNowResumesItsUsualSandlySelf.html.

WHY THE LUCKY STIFF. 2006. sandbox r50, here we go, loading conflicting gems. http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/8758.