Ruby master - Feature #8468
Remove $SAFE

06/01/2013 10:05 AM - shugo (Shugo Maeda)

<table>
<thead>
<tr>
<th>Status</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee</td>
<td>shugo (Shugo Maeda)</td>
</tr>
<tr>
<td>Target version</td>
<td>2.1.0</td>
</tr>
</tbody>
</table>

Description

Yesterday, at GitHub Tokyo drinkup (thanks, GitHub!), Matz agreed to remove the $SAFE == 4 feature from Ruby 2.1. Shibata-san, a developer of tDiary, which is the only application using $SAFE == 4, also agreed to remove it, so today is a good day to say goodbye to $SAFE (at least level 4).

Furthermore, I'm wondering whether $SAFE should be removed entirely, or not. Is there anyone using $SAFE?

Related issues:
- Related to Ruby master - Feature #15344: Being proactive about Ruby security
- Related to Ruby master - Feature #16131: Remove $SAFE, taint and trust

Feedback

Associated revisions

Revision 1f828497 - 06/12/2013 02:20 PM - shugo (Shugo Maeda)

- safe.c (rb_set_safe_level, safe_setter): raise an ArgumentError when $SAFE is set to 4. $SAFE=4 is now obsolete.
[ruby-core:55222] [Feature #8468]

- object.c (rb_obj_untrusted, rb_obj_untrust, rb_obj_trust):
  Kernel#untrusted?, untrust, and trust are now deprecated.
  Their behavior is same as tainted?, taint, and untaint, respectively.

- include/ruby/ruby.h (OBJ_UNTRUSTED, OBJ_UNTRUST): OBJ_UNTRUSTED() and OBJ_UNTRUST() are aliases of OBJ_TAINTED() and OBJ_TAINT(), respectively.

- array.c, class.c, debug.c, dir.c, encoding.c, error.c, eval.c,
  ext/curses/curses.c, ext/dbm/dbm.c, ext/dl/cfunc.c,
  ext/dl/cptr.c, ext/dl/dl.c, ext/etc/ctc.c, ext/fiddle/fiddle.c,
  ext/fiddle/pointer.c, ext/gdbm/gdbm.c, ext/readline/readline.c,
  ext/sdbm/init.c, ext/socket/anadata.c, ext/socket/basicsocket.c,
  ext/socket/socket.c, ext/socket/udpsocket.c,
  ext/stringio/stringio.c, ext/syslog/syslog.c, ext/tk/tcltklib.c,
  ext/win32ole/win32ole.c, file.c, gc.c, hash.c, io.c, iseq.c,
  load.c, marshal.c, object.c, proc.c, process.c, random.c, re.c,
  safe.c, string.c, thread.c, transcode.c, variable.c,
  vm_inshelper.c, vm_method.c, vm_trace.c: remove code for $SAFE=4.

- test/dl/test_dl2.rb, test/erb/test.erb.rb,
  test/readline/test_readline.rb,
  test/readline/test_readline_history.rb, test/ruby/test_alias.rb,
  test/ruby/test_array.rb, test/ruby/test_dir.rb,
  test/ruby/test_encoding.rb, test/ruby/test_env.rb,
  test/ruby/test_eval.rb, test/ruby/test_exception.rb,
  test/ruby/test_file_exhaustive.rb, test/ruby/test_hash.rb,
  test/ruby/test_io.rb, test/ruby/test_method.rb,
  test/ruby/test_module.rb, test/ruby/test_object.rb,
  test/ruby/test_pack.rb, test/ruby/test_rand.rb,
  test/ruby/test_regexp.rb, test/ruby/test_settracefunc.rb,
  test/ruby/test_struct.rb, test/ruby/test_thread.rb,
  test/ruby/test_time.rb: remove tests for $SAFE=4.
safe.c (rb_set_safe_level, safe_setter): raise an ArgumentError when $SAFE is set to 4. $SAFE=4 is now obsolete.
[ruby-core:55222] [Feature #8468]

object.c (rb_obj_untrusted, rb_obj_untrust, rb_obj_trust):
Kernel#untrusted?, untrust, and trust are now deprecated. Their behavior is same as tainted?, taint, and untaint, respectively.

include/ruby/ruby.h (OBJ_UNTRUSTED, OBJ_UNTRUST): OBJ_UNTRUSTED() and OBJ_UNTRUST() are aliases of OBJ_TAINTED() and OBJ_TAINT(), respectively.

array.c, class.c, debug.c, dir.c, encoding.c, error.c, eval.c, ext/curses/curses.c, ext/dbm/dbm.c, ext/dl/cfunc.c, ext/dl/cptr.c, ext/dl/dl.c, ext/etc/etc.c, ext/fiddle/fiddle.c, ext/fiddle/pointer.c, ext/gdbm/gdbm.c, ext/readline/readline.c, ext/sock/sock.c, ext/socket/ancdata.c, ext/socket/basicsocket.c, ext/socket/socket.c, ext/socket/udpsocket.c, ext/stringio/stringio.c, ext/syslog/syslog.c, ext/tk/tcltklib.c, ext/win32ole/win32ole.c, file.c, gc.c, hash.c, io.c, iseq.c, load.c, marshal.c, object.c, proc.c, process.c, random.c, re.c, safe.c, string.c, thread.c, transcode.c, variable.c, vm_inshelper.c, vm_method.c, vm_trace.c: remove code for $SAFE=4.

safe.c (rb_set_safe_level, safe_setter): raise an ArgumentError when $SAFE is set to 4. $SAFE=4 is now obsolete.

object.c (rb_obj_untrusted, rb_obj_untrust, rb_obj_trust): Kernel#untrusted?, untrust, and trust are now deprecated. Their behavior is same as tainted?, taint, and untaint, respectively.

include/ruby/ruby.h (OBJ_UNTRUSTED, OBJ_UNTRUST): OBJ_UNTRUSTED() and OBJ_UNTRUST() are aliases of OBJ_TAINTED() and OBJ_TAINT(), respectively.

Revision 41259 - 06/12/2013 02:20 PM - shugo (Shugo Maeda)

safe.c (rb_set_safe_level, safe_setter): raise an ArgumentError when $SAFE is set to 4. $SAFE=4 is now obsolete.

Revision 41259 - 06/12/2013 02:20 PM - shugo (Shugo Maeda)

safe.c (rb_set_safe_level, safe_setter): raise an ArgumentError when $SAFE is set to 4. $SAFE=4 is now obsolete.
object.c (rb_obj_untrusted, rb_obj_untrust, rb_obj_trust):
Kernel#untrusted?, untrust, and trust are now deprecated.
Their behavior is same as tainted?, taint, and untaint, respectively.

include/ruby/ruby.h (OBJ_UNTRUSTED, OBJ_UNTRUST):
OBJ_UNTRUSTED() and OBJ_UNTRUST() are aliases of OBJ_TAINTED() and OBJ_TAINT(), respectively.

array.c, class.c, debug.c, dir.c, encoding.c, error.c, eval.c,
ext/curses/curses.c, ext/dbm/dbm.c, ext/dl/cfunc.c,
ext/dl/cptr.c, ext/dl/dl.c, ext/etc/etc.c, ext/fiddle/fiddle.c,
ext/fiddle/pointer.c, ext/gdbm/gdbm.c, ext/readline/readline.c,
ext/sdbm/init.c, ext/socket/ancdata.c, ext/socket/basicsocket.c,
ext/socket/socket.c, ext/socket/udpsocket.c,
ext/stringio/stringio.c, ext/syslog/syslog.c, ext/tk/tcltklib.c,
ext/win32ole/win32ole.c, file.c, gcc.c, hash.c, io.c, iseq.c,
load.c, marshal.c, object.c, proc.c, process.c, random.c, re.c,
safe.c, string.c, thread.c, transcode.c, variable.c,
vm_insnhelper.c, vm_method.c, vm_trace.c: remove code for $SAFE=4.

test/dl/test_d2.rb, test/erb/test_erb.rb,
test/readline/test_readline.rb,
test/readline/test_readline_history.rb, test/ruby/test_alias.rb,
test/ruby/test_array.rb, test/ruby/test_dir.rb,
test/ruby/test_encoding.rb, test/ruby/test_env.rb,
test/ruby/test_eval.rb, test/ruby/test_exception.rb,
test/ruby/test_file_exhaustive.rb, test/ruby/test_hash.rb,
test/ruby/test_io.rb, test/ruby/test_method.rb,
test/ruby/test_module.rb, test/ruby/test_object.rb,
test/ruby/test_pack.rb, test/ruby/test_rand.rb,
test/ruby/test_regexp.rb, test/ruby/test_settracefunc.rb,
test/ruby/test_struct.rb, test/ruby/test_thread.rb,
test/ruby/test_time.rb: remove tests for $SAFE=4.

Revision 41259 - 06/12/2013 02:20 PM - shugo (Shugo Maeda)

safe.c (rb_set_safe_level, safe_setter): raise an ArgumentError
when $SAFE is set to 4. $SAFE=4 is now obsolete.
[ruby-core:55522] [Feature #8468]

object.c (rb_obj_untrusted, rb_obj_untrust, rb_obj_trust):
Kernel#untrusted?, untrust, and trust are now deprecated.
Their behavior is same as tainted?, taint, and untaint, respectively.

include/ruby/ruby.h (OBJ_UNTRUSTED, OBJ_UNTRUST):
OBJ_UNTRUSTED() and OBJ_UNTRUST() are aliases of OBJ_TAINTED() and OBJ_TAINT(), respectively.

array.c, class.c, debug.c, dir.c, encoding.c, error.c, eval.c,
ext/curses/curses.c, ext/dbm/dbm.c, ext/dl/cfunc.c,
ext/dl/cptr.c, ext/dl/dl.c, ext/etc/etc.c, ext/fiddle/fiddle.c,
ext/fiddle/pointer.c, ext/gdbm/gdbm.c, ext/readline/readline.c,
ext/sdbm/init.c, ext/socket/ancdata.c, ext/socket/basicsocket.c,
ext/socket/socket.c, ext/socket/udpsocket.c,
ext/stringio/stringio.c, ext/syslog/syslog.c, ext/tk/tcltklib.c,
ext/win32ole/win32ole.c, file.c, gcc.c, hash.c, io.c, iseq.c,
load.c, marshal.c, object.c, proc.c, process.c, random.c, re.c,
safe.c, string.c, thread.c, transcode.c, variable.c,
vm_insnhelper.c, vm_method.c, vm_trace.c: remove code for $SAFE=4.

test/dl/test_d2.rb, test/erb/test_erb.rb,
test/readline/test_readline.rb,
Revision 41259 - 06/12/2013 02:20 PM - shugo (Shugo Maeda)

- safe.c (rb_set_safe_level, safe_setter): raise an ArgumentError when $SAFE is set to 4. $SAFE=4 is now obsolete.
  [ruby-core:55222] [Feature #8468]

- object.c (rb_obj_untrusted, rb_obj_untrust, rb_obj_trust):
  Kernel#untrusted?, untrust, and trust are now deprecated. Their behavior is same as tainted?, taint, and untaint, respectively.

- include/ruby/ruby.h (OBJ_UNTRUSTED, OBJ_UNTRUST): OBJ_UNTRUSTED() and OBJ_UNTRUST() are aliases of OBJ_TAINTED() and OBJ_TAINT(), respectively.

- array.c, class.c, debug.c, dir.c, encoding.c, error.c, eval.c, ext/curses/curses.c, ext/dbm/dbm.c, ext/dl/cfunc.c, ext/dl/cptr.c, ext/dl/dl.c, ext/etc/etc.c, ext/fiddle/fiddle.c, ext/fiddle/pointer.c, ext/gdbm/gdbm.c, ext/readline/readline.c, ext/sdbm/init.c, ext/socket/anodat.c, ext/socket/basicsocket.c, ext/socket/socket.c, ext/socket/udpsocket.c, ext/stringio/stringio.c, ext/syslog/syslog.c, ext/tk/tcltklib.c, ext/win32ole/win32ole.c, file.c, gc.c, hash.c, io.c, icense.c, load.c, marshal.c, object.c, proc.c, process.c, random.c, re.c, safe.c, string.c, thread.c, transcode.c, variable.c, vm_insnhelper.c, vm_method.c, vm_trace.c: remove code for $SAFE=4.


History

#1 - 06/01/2013 10:36 AM - headius (Charles Nutter)

JRuby has not supported $SAFE for a long time, and the only complaints we've heard are from people that thought they needed $SAFE (and did not) or people who just happened to notice it was missing.

We strongly support removing $SAFE, but I would like to help come up with a better (more fine-grained, more configurable, more reliable) security mechanism to replace it as well.

#2 - 06/01/2013 12:38 PM - Student (Nathan Zook)

One wonders where everyone has been for the last five months. The entire Rails security fiasco would have been avoided if DHH had seen fit to write Rails with $SAFE = 1. This is one thing that the Perl community got right almost immediately which so far seems to have completely evaded the Ruby community. External data is NOT to be trusted, and $SAFE = 0 is only appropriate for script which are ENTIRELY isolated from the outside.

Yes, there is the hole with symbols, but since symbol creation with outside data is a memory leak anyway, the correct thing to do would be to forbid
Now, if you have a superior solution to $SAFE, I am all ears—especially since the community hasn't seen fit to understand and use it.

#3 - 06/01/2013 07:04 PM - headius (Charles Nutter)

$SAFE is not a good security option for at least a few reasons:

- It requires maintaining checks for both tainting and safe levels in nearly every piece of C code attached to Ruby, including extensions. This is both a maintenance and performance nightmare.
- It provides a very coarse-grained security, where many secured features are only secured at levels that prevent most applications from working at all (due to other secured features being needed).
- It is blacklisting, which is almost impossible to do without leaving gaps. Every new API needs to enlist in the blacklisting, every change needs to be aware of it, and if you don't choose the right safe level or one piece of code isn't aware of it, you've got a hole.

The security model provided on the JVM or on operating systems with access control lists are both better options. If you run with security on, everything is forbidden; you must explicitly turn on the permissions you want and whitelist those capabilities. Those permissions are fine-grained, allowing you to disable only code evaluation or filesystem access or dynamic library loading, rather than having to choose from four pre-determined blacklists.

Regarding the Rails exploit...SAFE=1 may or may not have helped, but the real problem was allowing arbitrary code to be embedded and executed from a data format in the first place.

Regarding Perl...even the Perl folks say tainting is not intended to be used as a general security mechanism...it is at best a way to audit code for egregious security flaws.

Regarding why the Ruby community has not moved away from SAFE...they actually have. For most online servers that are used to run user code (tryruby.org, various online ruby tutorials and schools, etc) they are using JRuby with JVM security policies rather than the broken and troublesome SAFE support in MRI. It's time for MRI to move away from SAFE as its sole security model too.

#4 - 06/02/2013 06:26 AM - Anonymous

@Nathan: Do you mean that Perl has $SAFE = 1 by default?

#5 - 06/02/2013 12:09 PM - spatulasnout (B Kelly)

shugo (Shugo Maeda) wrote:

Shibata-san, a developer of tDiary, which is the only application using $SAFE == 4, also agreed to remove it, so today is a good day to say goodbye to $SAFE (at least level 4).

For the record, our C++ application embeds ruby (currently both 1.8.4 and 1.9.3 interpreters), and we've been using $SAFE == 4 since 2006 or so.

As I'd described in [ruby-core:36950]--

We use $SAFE = 4 to create a sandbox for the execution of semi-trusted scripts, conforming to our application's plug-in API.

"Semi-trusted" meaning of third-party origin, but not intentionally malicious.

(I wouldn't bet anything valuable that our $SAFE = 4 sandbox could contain a maliciously coded script intent on breaking out of it.)

The specific case we're guarding against is a well-intentioned but buggy third-party plug-in, which, when installed by one of our users and executed by our application, might end up destroying data on the user's filesystem.

The $SAFE = 4 sandbox only allows I/O through our plug-in API, which restricts I/O to only the set of subdirectories and/or files that are pertinent to the operation being requested of the plug-in.
Note, I'm not particularly wedded to the $SAFE security model. (I have used $SAFE = 1 in a few web/CGI scripts.)

What I'd really like is a mechanism in Ruby that would provide a secure sandbox that could contain completely untrusted code.

Regards,

Bill

#6 - 06/02/2013 03:57 PM - Student (Nathan Zook)

boris_stitnicky (Boris Stitnicky) wrote:

    @Nathan: Do you mean that Perl has $SAFE = 1 by default?

No, I'm saying that perl has a taint property very much like ruby's, and perl has a safe mode very similar to ruby's $SAFE = 1, and that the perl community takes it seriously.

The result is that the nonsense which we hit in Jan hasn't been a problem there for a long, long time. I'm not saying that their are no security flaws in perl programs, far from it. I'm saying that trusting user input by default in a web app in the 21st century is a special kind of stupid.

#7 - 06/02/2013 04:47 PM - Student (Nathan Zook)

headius (Charles Nutter) wrote:

    $SAFE is not a good security option for at least a few reasons:

    - It requires maintaining checks for both tainting and safe levels in nearly every piece of C code attached to Ruby, including extensions. This is both a maintenance and performance nightmare.

    Part of the issue here is the balance of pain. You have the pain of the core team verses the pain of platform providers verses the pain of end developers verse the pain of the organizations that use the code in question. We now have a Rails server botnet thanks to $SAFE = 0. How much pain is that going to be in the world?

    Do you have specifics about the performance costs?

    - It provides a very coarse-grained security, where many secured features are only secured at levels that prevent most applications from working at all (due to other secured features being needed.

    I have a real hard time with your vagueness here. All SAFE=1 does is prevent tainted data from being executed, for a broad definition of executed. Unless you are doing something like tryruby.org, there is no reason at all that this should ever be a problem. If you are, then of course, you are going to need a more robust security model than SAFE=1 can provide.

    - It is blacklisting, which is almost impossible to do without leaving gaps. Every new API needs to enlist in the blacklisting, every change needs to be aware of it, and if you don't choose the right safe level or one piece of code isn't aware of it, you've got a hole.

    I agree that blacklisting is problematic in general. I don't agree that it is nearly so dire at this point as you say. As I mentioned, SAFE=1 is solely intended to prevent tainted strings from being executed, in the broad meaning of the term. Defined in this way, it is pretty straight forward to reason about. If you aren't mucking around with some new form of I/O, set SAFE=1, don't be stupid about untainting, and be happy. If you are, well, then you **** sure better dot your i's & cross your t's. Security is just one of the things on that list.

    I don't know how the implementation is done, but it seems to me that internally, it is all about keeping up with the taint bit, and that it is only at the edges (I/O & eval) that $SAFE=1 comes into play.

    The security model provided on the JVM or on operating systems with access control lists are both better options. If you run with security on, everything is forbidden; you must explicitly turn on the permissions you want and whitelist those capabilities. Those permissions are fine-grained, allowing you to disable only code evaluation or filesystem access or dynamic library loading, rather than having to choose from four pre-determined blacklists.

    NOW we have an alternative suggestion. But I'm not thrilled with what you are suggesting. Perl's safe mode is very close to our $SAFE=1, and they make use of it. We don't. Now you suggest requiring end developers, who can't be bothered to worry about which of 5 options to use, to make a dozen or so correct fine-grained choices on things they might not understand at all?

    In real life, this looks something like the following: Write code. See code fail. See the security mechanism. Disable the security mechanism. Repeat. This mentality regarding SAFE is already here. Given the response to bug January, I don't expect things to get better by making devs work more to achieve the result.
Regarding the Rails exploit...SAFE=1 may or may not have helped, but the real problem was allowing arbitrary code to be embedded and executed from a data format in the first place.

I would say that this is exactly what SAFE=1 is supposed to prevent. Those strings are tainted, and under SAFE=1, you can't eval tainted strings.

The fact that Psyche calls []= on objects whenever the user-supplied data tells it to is a problem in its own right, but even that would not have been a problem if SAFE=1.

The symbol exploit probably would still be there. My Symbol[] proposal is designed to allow an easy fix for that one.

Regarding Perl... even the Perl folks say tainting is not intended to be used as a general security mechanism...it is at best a way to audit code for egregious security flaws.

Regarding why the Ruby community has not moved away from SAFE... they actually have. For most online servers that are used to run user code (tryruby.org, various online ruby tutorials and schools, etc) they are using JRuby with JVM security policies rather than the broken and troublesome SAFE support in MRI. It's time for MRI to move away from SAFE as its sole security model too.

Unlike the originator of this thread, you at least appear to be proposing that we move TO something. Certainly having fine-grained control available is desirable. I will argue, however, that adoption goes much better if you have one or two big red buttons to push instead of a control board full of sliders. I will look into the JVM security model & see what I think.

#8 - 06/02/2013 09:15 PM - spatulasnout (B Kelly)

spatulasnout (B Kelly) wrote:

What I'd really like is a mechanism in ruby
that would provide a secure sandbox that could contain completely untrusted code.

To clarify:

I'd say our application treats the difference between $SAFE == 4 and $SAFE <= 1 as somewhat similar to the boundary between 'privileged mode' and 'user mode' on a CPU.

The functionality on which we depend includes:

- user-mode (sandboxed) code is restricted:
  - can't modify/extend existing classes & modules
  - can't perform I/O
  - can't modify global or thread-local variables or environment

- user-mode code MAY transition to privileged-mode via a "trap" (i.e. calling a block that was compiled in a privileged context.)

Now, if something like the JVM provides a more granular permission structure, that sounds potentially great.

But if we could at least just begin with the most restrictive possible sandbox, and then allow a trap to a privileged context (via calling a proc compiled at a privileged level,) my hope is such a thing could be accomplished without some of the complexity attributed to current $SAFE handling:

headius (Charles Nutter) wrote:

[$SAFE] is blacklisting, which is almost impossible to do without leaving gaps. Every new API needs to enlist in the blacklisting, every change needs to be aware of it, and if you don't choose the right safe level or one piece of code isn't aware of it, you've got a hole.

It would indeed seem ideal to require, for any Ruby VM, that the default for any newly defined...
native extension function is to enforce requiring privileged execution -- such that the simplest path for anyone coding a ruby extension is to define functions that the VM will only execute in a privileged context.

It shouldn't be that "every new API" needs to enlist in the blacklisting. Every new pure Ruby API shouldn't need to do anything; it's either being executed in a privileged context or it isn't.

It's only the native extensions that need to be specially attended to; and if they can default to requiring privileged execution unless explicitly marked as sandbox-safe by the author, then it seems to me we'd be in pretty good shape?

Regards,
Bill

#9 - 06/02/2013 10:30 PM - charliesome (Charlie Somerville)

Personally, I don't think Ruby should have any security features like $SAFE or granular sandboxing at all.

Trying to build a sandboxed environment for running untrusted code at the same level as the code being run is a fundamentally flawed idea. If a $SAFE check is missed in just one place - the entire security of the system comes tumbling down.

Granular sandboxing like the JVM provides is also flawed in the same way as $SAFE. Sure, it might be marginally better in some ways, but it still shares the same weakness as $SAFE - one missing check anywhere in the system (whether in CRuby itself, or a C extension) compromises the security of the entire system. You only need to look at all the Java 0-days to see how a single tiny mistake made in privileged code can lead to total compromise.

The real solution to this problem is to run a sandbox at the level above Ruby. I run untrusted code on http://eval.in inside a ptrace based sandbox. System calls must be explicitly whitelisted, and strict checks are performed on arguments before allowing a system call to be made. This significantly reduces the attack surface to just a handful of system calls, which is far better than trying to make sure that every single part of a massive programming language like Ruby is secure.

Instead of encouraging people to rely on flawed/broken security models like $SAFE or VM-level granular sandboxing, we should encourage them to rely on a more robust OS-level solution instead.

I would be quite happy to see features like $SAFE, trust, and taint removed from the Ruby programming language.

#10 - 06/02/2013 10:42 PM - headius (Charles Nutter)

Student (Nathan Zook) wrote:

Part of the issue here is the balance of pain. You have the pain of the core team verses the pain of platform providers verses the pain of end developers verse the pain of the organizations that use the code in question. We now have a Rails server botnet thanks to $SAFE = 0. How much pain is that going to be in the world?

I'm not arguing that there shouldn't be any security system at all...I'm just arguing that the current coarse-grained, blacklisting system is too flawed to be the model we follow.

Do you have specifics about the performance costs?

I have attempted to remove tainting checks in the pasts, and in some cases the performance of small operations improved by 10-15%. Now of course most operations that are not creating new objects or mutating existing ones don't hit this, but it's a pretty big hit when it's required.

JRuby has been systematically removing our broken implementation of $SAFE level checks as well, and it's always an improvement in performance.

• It provides a very coarse-grained security, where many secured features are only secured at levels that prevent most applications from working at all (due to other secured features being needed.

I have a real hard time with your vagueness here. All SAFE=1 does is prevent tainted data from being executed, for a broad definition of executed. Unless you are doing something like tryruby.org, there is no reason at all that this should ever be a problem. If you are, then of course, you are going to need a more robust security model than SAFE=1 can provide.

The problem is that you have to really, really trust that strings are not getting untainted incorrectly, and that all possible paths for user strings are tainting those strings. It's a bad model, and one bad apple spoils the whole thing.
I don't know how the implementation is done, but it seems to me that internally, it is all about keeping up with the taint bit, and that it is only at the edges (I/O & eval) that $SAFE=1 comes into play.

Yes. Keeping up with the taint bit in some thousands of lines of code that must properly propagate it, with very sparse tests. Bad model.

Security should not be a characteristic of objects in the system but of execution contexts. A given thread should either be able to evaluate new code or not be able. Allowing applications to get around that hard limitation by gaming tainting is just asking for trouble. Enable or disable specific privileges, and be done with it. NEVER trust any object and allow it to skip security, ever.

The security model provided on the JVM or on operating systems with access control lists are both better options. If you run with security on, everything is forbidden; you must explicitly turn on the permissions you want and whitelist those capabilities. Those permissions are fine-grained, allowing you to disable only code evaluation or filesystem access or dynamic library loading, rather than having to choose from four pre-determined blacklists.

NOW we have an alternative suggestion. But I'm not thrilled with what you are suggesting. Perl's safe mode is very close to our $SAFE=1, and they make use of it. We don't. Now you suggest requiring end developers, who can't be bothered to worry about which of 5 options to use, to make a dozen or so correct fine-grained choices on things they might not understand at all?

In real life, this looks something like the following: Write code. See code fail. See the security mechanism. Disable the security mechanism. Repeat. This mentality regarding SAFE is already here. Given the response to bug January, I don't expect things to get better by making devs work more to achieve the result.

Although I didn't write it up here, I think it would be simple to provide the existing SAFE levels as pre-defined sets of permissions on e.g. JVM. The weirdness would be mapping the safe levels' blacklists to security policy's whitelists, but it's doable. So we could have a set of pre-existing new whitelist-based SAFE level policies (with explicit configurations), but users could mix and match as desired. So you get your big red buttons, but many (most?) users will customize appropriate to their needs.

Regarding the Rails exploit...SAFE=1 may or may not have helped, but the real problem was allowing arbitrary code to be embedded and executed from a data format in the first place.

I would say that this is exactly what SAFE=1 is supposed to prevent. Those strings are tainted, and under SAFE=1, you can't eval tainted strings.

The fact that Psyche calls []= on objects whenever the user-supplied data tells it to is a problem in its own right, but even that would not have been a problem if SAFE=1.

If tainting is propagating correctly. Again, implicitly trusting "blessed" user-mutable objects/data is always wrong.

The symbol exploit probably would still be there. My Symbol[] proposal is designed to allow an easy fix for that one.

The symbol exploit just needs symbols to be GCable. I've wanted to do this for a while in JRuby, so I think we'll just fix it...and I believe MRI is probably going to fix it too soon too.

#11 - 06/02/2013 10:47 PM - headius (Charles Nutter)

A path forward to defining a more fine-grained security model...

- Define in clear terms what we want to restrict. The existing safe levels would be a quick way to start doing this, since we already have a list of things that are (dis)allowed at particular levels.

- Put together a simple configuration or command-line format that allows assembling those permissions into security policies. As a bonus feature, reimplement current SAFE levels by simply selecting one of a pre-defined set of policies that roughly maps to current SAFE level restrictions.

I've done bits and pieces of this for JRuby, but this bug would probably be a good forum to formalize it. The API part of this is actually very small, and the implementation part is also not particularly large, since many of the checks are already in place for SAFE levels. We just need to flip them to positive checks (can I do this) rather than negative checks (is this disallowed) and eliminate the idea that untainting or trusting objects can get around security.

Looking forward to working on this.

#12 - 06/03/2013 05:24 PM - jballanc (Joshua Ballanco)

I agree with charliesome that sandboxing is best handled by the running system, and not as a language feature. Still, I think we can include support for such facilities (where they are available) in a standardized way. Might I suggest something along the lines of MacRuby's Sandbox or the RubyGem Playpen (https://github.com/tenderlove/playpen)? The general idea would be to have a few predefined, high-level sandbox directives (no_disk, no_internet, etc.), as well as a defined mechanism for accessing lower-level facilities unique to each platform.

While this is obviously not an ideal solution, I fear it is the best we could do without re-inventing Ruby from the ground-up with sandboxing in mind.
(perhaps mRuby will have better luck in this regard). Finally, it might be useful to consider the discussion/debate the Clojure community recently went through with regards to read-eval (a discussion that took place in response to the recent Rails/YAML vulnerabilities), as well as the ultimate conclusion that it is futile to attempt, in effect, to secure "eval": https://groups.google.com/d/topic/clojure-dev/zG90eRnbbJQ/discussion

#13 - 06/06/2013 08:29 PM - arton (Akio Tajima)

Hi,

I use $SAFE in ActiveScriptRuby (IActiveScript interface for ruby) for implementing IObjectSafety interface (Ixxx is defined by Component Object Model specification but I think it's not your concern so I don't describe it).

My living code is "rb_safe_level() >= 3", so it's ok to remove level 4 but I'm not pleased to remove it completely.

Regards

--
aronx@yahoo.co.jp

#14 - 06/10/2013 01:46 PM - Student (Nathan Zook)

jballanc (Joshua Ballanco) wrote:

... Clojure community recently went through with regards to read-eval (a discussion that took place in response to the recent Rails/YAML vulnerabilities), as well as the ultimate conclusion that it is futile to attempt, in effect, to secure "eval": https://groups.google.com/d/topic/clojure-dev/zG90eRnbbJQ/discussion

I don't think anyone here is trying to secure "eval", unless by "secure" you mean "prevent it from being called on untrusted data". The discussion is about how to express this in a clean & performant fashion.

#15 - 06/10/2013 05:56 PM - shugo (Shugo Maeda)

- Status changed from Feedback to Assigned
- Assignee changed from shugo (Shugo Maeda) to matz (Yukihiro Matsumoto)

Thanks for your feedback, guys.

It's OK for me to leave $SAFE < 3 because it's just a fail-safe feature. However, safe level 4 is considered harmful, because it provides a pseudo sandbox, which is vulnerable by design. People tend to expect it as a real sandbox, but it really isn't.

I propose two options for Ruby 2.1:

1. Raise an error when $SAFE is set to 4.
2. Show a warning like "Safe level 4 is deprecated. It provides a pseudo sandbox, which can be used only for semi-trusted code." when $SAFE is set to 4, and keep the current behavior.

What do you think, Matz?

In either case, I wouldn't like to address vulnerabilities in $SAFE anymore. Some people seem to believe I'm the maintainer of $SAFE, but I'm not. I has addressed vulnerabilities in $SAFE just because other people haven't.

#16 - 06/10/2013 06:43 PM - matz (Yukihiro Matsumoto)

I agree with Shugo's plan. Shugo, could you make it raise exceptions to keep you away from worrying on $SAFE any longer?

Matz.

#17 - 06/10/2013 06:51 PM - shugo (Shugo Maeda)

- Assignee changed from matz (Yukihiro Matsumoto) to shugo (Shugo Maeda)

matz (Yukihiro Matsumoto) wrote:

I agree with Shugo's plan. Shugo, could you make it raise exceptions to keep you away from worrying on $SAFE any longer?

I'll do it. Thanks.

#18 - 06/10/2013 08:50 PM - headius (Charles Nutter)
So at this point, the only thing that is being removed is the sandboxing provided by $SAFE=4, correct?

I should note that my concerns about using $SAFE as a security mechanism (coupled with taint/untrust) is still just as prone to holes as ever, with or without sandboxing. Blacklisting systems...especially where you are blacklisting against data bits modifiable outside the security flow of the application, are inherently untrustworthy.

#19 - 06/10/2013 09:09 PM - shugo (Shugo Maeda)

headius (Charles Nutter) wrote:

So at this point, the only thing that is being removed is the sandboxing provided by $SAFE=4, correct?

Yes.

I should note that my concerns about using $SAFE as a security mechanism (coupled with taint/untrust) is still just as prone to holes as ever, with or without sandboxing. Blacklisting systems...especially where you are blacklisting against data bits modifiable outside the security flow of the application, are inherently untrustworthy.

$SAFE < 4 is a fail-safe feature to detect bugs of applications, so it's not so serious that it's untrustworthy, which just means some bugs are detected by $SAFE but other bugs are not.

However, I doubt it's worthwhile.

#20 - 06/12/2013 11:20 PM - shugo (Shugo Maeda)

- Status changed from Assigned to Closed
- % Done changed from 0 to 100

This issue was solved with changeset r41259.

Shugo, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

- safe.c (rb_set_safe_level, safe_setter): raise an ArgumentError when $SAFE is set to 4. $SAFE=4 is now obsolete. [ruby-core:55222] [Feature #8468]

- object.c (rb_obj_untrusted, rb_obj_untrust, rb_obj_trust):
  Kernel#untrusted?, untrust, and trust are now deprecated. Their behavior is same as tainted?, taint, and untaint, respectively.

- include/ruby/ruby.h (OBJ_UNTRUSTED, OBJ_UNTRUSTED()): OBJ_UNTRUSTED() and OBJ_UNTRUST() are aliases of OBJ_TAINTED() and OBJ_TAINT(), respectively.

- array.c, class.c, debug.c, dir.c, encoding.c, error.c, eval.c, ext/curses/curses.c, ext/dbm/dbm.c, ext/dl/cfunc.c, ext/dl/cptr.c, ext/dl/dl.c, ext/etc/etc.c, ext/fiddle/fiddle.c, ext/fiddle/pointer.c, ext/gdbm/gdbm.c, ext/reading/reading.c, ext/sdbm/init.c, ext/socket/ancdata.c, ext/socket/basicsocket.c, ext/socket/socket.c, ext/socket/udpsocket.c, ext/stringio/stringio.c, ext/syslog/syslog.c, ext/tk/cTklib.c, ext/win32ole/win32ole.c, file.c, gc.c, hash.c, io.c, iseq.c, load.c, marshal.c, object.c, proc.c, process.c, random.c, re.c, safe.c, string.c, thread.c, transcode.c, variable.c, vm_insins.c, vm_method.c, vm_trace.c: remove code for $SAFE=4.

#21 - 11/27/2018 03:07 AM - shyouhei (Shyouhei Urabe)
- Related to Feature #15344: Being proactive about Ruby security added

#22 - 08/29/2019 07:16 AM - naruse (Yui NARUSE)
- Related to Feature #16131: Remove $SAFE, taint and trust added