A Paranoid Perspective of an Interpreted Language

Dominique Brezinski
Black Hat, Inc.

Black Hat Briefings

Goals

Review several vulnerabilities in the Ruby language implementation to better understand what vulnerabilities in high-level languages look like

Understand what is different about auditing software written in high-level, interpreted languages versus C/C++

High-level Programming Languages

We are focusing on issues associated with the use of interpreted, object-oriented languages like Ruby, Python, C# and Smalltalk, though many of the issues apply equally to other interpreted languages like Perl.

Ruby is used for all the examples, but I am not picking on Ruby specifically--I just like Ruby and use it all the time. The issues presented here apply to all similar languages, though the bugs shown are specific to Ruby.

Why?

Even though developers do not have to worry about memory manipulation and potential buffer overflows directly in their code when writing in higher-level languages, using higher-level languages alone does not mean the resulting software will be secure. There are still many ways to shoot yourself in the foot when writing software in high-level languages, as well as deep dependencies in libraries and interpreters that can be attacked under certain conditions.

Interpreter = a complex piece of software

- The interpreters for most of the languages are fairly complex pieces of software, generally written in C
- The standard libraries, which are what make many of these languages very efficient for common programming tasks, are also generally written in C
- The developers of language interpreters are better than average developers, but that does not mean there are no mistakes

Example: pack.c

```
static void
encodes(VALUE  str,  char *  s,  long len,  int type)
{
    char *buff = ALLOCA_N(char, len * 4 / 3 + 6); /* len > 1GB will cause int overflow and small
allocation */
    long i = 0;
    char *trans = type == 'u' ? uu_table : b64_table;
    int padding;

    if (type == 'u') {
        buff[i++] = len + ' ';
        padding = '`';
    }
    else {
        padding = '=';
    }
    while (len >= 3) { /* bounded by len, but probably not exploitable due to stack memory
constraints */
        buff[i++] = trans[077 & (*s >> 2)];
        buff[i++] = trans[077 & (((*s << 4) & 060) | ((s[1] >> 4) & 017))];
        buff[i++] = trans[077 & (((s[1] << 2) & 074) | ((s[2] >> 6) & 03))];
        buff[i++] = trans[077 & s[2]];
        s += 3;
        len -= 3;
    }...
```

Example: array.c (found by John McDonald)

```
    case 3:
     [snipped]
        len = NIL_P(arg2) ? RARRAY(ary)->len - beg : NUM2LONG(arg2); /* len is from method argument */
        break;
    }
    rb_ary_modify(ary);
    end = beg + len; /* end is derived from len */
    if (end > RARRAY(ary)->len) { /* when a long len is specified, this will be true */
        if (end >= RARRAY(ary)->aux.capa) { /* this will be true also */
            REALLOC_N(RARRAY(ary)->ptr, VALUE, end); /* size passed to realloc() is sizeof(VALUE) *
end--WRAP*/
            RARRAY(ary)->aux.capa = end;
        }
[snipped]
   if (block_p) {
[snipped]
   }
   else {
        p = RARRAY(ary)->ptr + beg;
        pend = p + len; /* derived from len, which is now much longer than memory allocated */
        while (p < pend) {
            *p++ = item; /* HEAP OVERFLOW! */
        }
    }
    return ary;
}
```

Lesson One

The interpreters and binary libraries need to be audited just like any other sensitive program. Unlike most programs, all input should be viewed as potentially hostile, since the code being evaluated could be untrusted.

Note: this audit is separate from verifying that the language implementation is correct.

Example: Breaking Safe (example by Matz)

```
#!/usr/bin/env ruby
def safe_eval(str)
  Thread.start {
    $SAFE=4
    eval str
  }.value
 end
begin
  safe_eval("puts :foo")   #=> security error--no direct output allowed when $SAFE == 4
rescue
  puts "Caught exception"
end
result = safe_eval(<<-END)
   o = Object.new
   def o.to_s  #=> singleton method to_s, which is not being marked TAINTED
      puts :foo
   end
   o
END
puts result #=> o.to_s gets called by puts when object is not a string
```

Lesson Two

Execution restrictions within languages are good features but are very difficult to implement correctly. It takes time and very knowledgeable people reviewing the implementations before such features can be trusted. If an application needs a high level of security, trusting semantic restrictions in languages to be the basis of the application security is not wise. Use them, but do not *rely* on the security of the feature. This holds true for Ruby's safe levels, Perl's Safe.pm, and all the rest.

Example: XML-RPC (publicly disclosed)

- Any application that provided XML-RPC functionality using XMLRPC.iPIMethods allowed remote command execution.
- The source of the vulnerability was included methods exported from ***all*** ancestors

Example Continued

The diff of the fix:

```
--- ruby-1.8.2/lib/xmlrpc/utils.rb.orig 2003-08-15 02:20:14.000000000 +0900
+++ ruby-1.8.2/lib/xmlrpc/utils.rb      2005-07-01 16:33:19.243521736 +0900
@@ -138,7 +138,7 @@

     def get_methods(obj, delim=".")
       prefix = @prefix + delim
-      obj.class.public_instance_methods.collect { |name|
+      obj.class.public_instance_methods(false).collect { |name|
         [prefix + name, obj.method(name).to_proc, nil, nil]
       }
     end
```

Example: server.rb

Exception handling is a great language feature, but it is also easy to forget to reset aspects of program state when exceptions are caught. Also, understanding what and when exceptions can be thrown is not always easy. A common mistake is to put in a catch-all exception handler with only the expectation that certain classes of exceptions will occur. Then, unbeknownst to the developer, a library used will raise an exception of a different class that should not have been caught locally, that will get caught, and program state will change in an unexpected way. Security issues can result.

Lesson Three

Building on other code, either through OO inheritance or standard libraries, is good for many reasons, but it is also a potential source of many security problems. There may be functionality present in the class ancestors or library that *must not* be included in the application and needs to be overridden or excluded, but the developer needs to know the functionality is present in the first place. Likewise, exception raised in libraries need to be well understood. Audits often need to extend into the programs dependencies, such as standard and third-party libraries.

Automated Static Source Analysis

- Not nearly as useful for interpreted languages, since implementation security issues tend to be logic flaws, code *exclusion* or execution-time dependent rather than dangerous API usage etc.
- Interpreted languages tend to have more functional density per kloc, therefore less code per application, so manual review is more feasible and higher return
- Modified dependency analyzers and class browsers can be useful tools for directing manual reviews

Binary Analysis

- Can (should) be used on interpreter and binary libraries
- No binary for the application developed in the interpreted language, so binary analysis is not applicable to the end application

Dynamic Analysis

- It is possible to use the introspective features of these languages to list key information like public methods of classes and other security relevant information during execution
- Build mix-in modules that can provide generic analysis (class attributes and methods), while others would be specific to the application (similar to debug-only code)
- Could probably be based off of existing developer tools

Manual Code Review

Code review is probably the best tool we currently have to evaluate the security of an application written in an interpreted language. Defining a process to determine the scope of the code review is beneficial.

Review Scope

- Determine library dependencies, so they can be included in the audit
- Determine the class lineage (inheritance) for all classes used in the application, so the parent classes and the inherited functionality can be included in the audit
- As the review progresses, certain portions of the interpreter should be included in the audit based on the language features used (i.e. reflection)

Pitfalls of Inheritance

Having a feature-rich set of base classes greatly decreases the amount of code a developer needs to write to implement a given set of features, since many features can be implemented by creating new classes that inherit a great deal of functionality. However, the resulting code will possibly implement *more* functionality that is not secure. It is often necessary to over-ride unnecessary and/or unwanted functionality inherited from parent classes. Remember the XML-RPC vulnerability? In programs that make use of OO language features, auditors should verify only necessary functionality is inherited from parent classes.

Reflection Point One

Reflection is the ability to dynamically change the structure of the executing program, for-instance, by adding methods to a class based on the execution environment, configuration changes, and/or user input. Understanding when and how reflection is used in a program can be important from a security perspective. For instance, are reflectively created method names derived from attacker controllable input? If so, are protections in place to ensure that existing, sensitive methods cannot be redefined due to malicious input?

Reflection Point Two

Normally many parts of the interpreter are not subject to abuse by a user of the program. If code is dynamically generated or changed based off of user input, then normally unaccessible parts of the interpreter could be subject to attack (i.e. parser, introspective methods).

Sandboxes for Untrusted Code

Any attempts to execute untrusted code in sandboxes should be examined *very* carefully. If an attacker can generate code that gets executed in a sandbox, both ***language implementation of the sand-boxing mechanisms*** and ***interpreter implementation*** are ***subject to attack***. Earlier examples demonstrated problems with both language security mechanism and interpreter implementation. There will be more. If a program relies on sandboxes for safety, then the interpreter code needs to be included in the audit.

What does this all mean?

- Programs written in all languages can have exploitable issues.
- While programming at a higher level of abstraction removes certain lower level vulnerabilities from the general case, it does not mean the vulnerabilities are completely eradicated
- Programs that need very high levels of security cannot rely on the implementation language to deliver assurance
- We need more knowledge and better analysis tools for doing security audits of programs written in interpreted languages