Virtual Meta-Scripting

Bytecode for PHP and JavaScript

Ben Fuhrmannek
bef@sektioneins.de

May 2010

Abstract

Both PHP and JavaScript are frequently being targeted for exploiting web applications. This article elaborates on the idea of building a set of virtual machines on top of each programming language. As a result a single type of bytecode can be executed by both VMs. Particular emphasis is put on designing virtual machines to be most suitable for code obfuscation in a post exploitation scenario.

1 Introduction

The combination of JavaScript and PHP is widely used[1] for building web applications - where PHP covers the server side and JavaScript is primarily known as a client side programming language. This article elaborates the idea of executing a single type of bytecode within virtual machines implemented in both JavaScript and PHP. This could be useful for optimising the development process of web applications, since one programming language and one compiler were able to generate all client and server side components. However, throughout this article - and for the purpose of stimulating the reader's imagination - a post-exploitation scenario is assumed. Bytecode and virtual machines serve as means to code obfuscation. The post-exploitation scenario denotes either of two possibilities, or both: (1) An attacker tries to conceal the attack vector for exploiting a vulnerable web application by injecting a virtual machine and further payload encoded appropriately. (2) An attacker has already exploited a web application and tries to establish a more permanent basis for further practical research.

But before going any further, it appears best to define the terminology used:

- *Obfuscation* in the context of bytecode interpretation refers to the transformation of a program in such a way, that reverse engineering the program is more uncomfortable and less automated as compared to the untransformed program, while keeping the program's function intact.

[1] Websites for programming language popularity[3, 4] clearly show PHP and JavaScript within the top ten most mentioned languages as of March 2010. In addition, the following assumptions were made: (1) Searches or chatter correlates with overall use or popularity of a programming language. (2) The general preference for a language correlates with the language choice for a web application. (3) The popularity of two languages used in combination correlates with the product of their respective probability of being used. It can now be concluded, that the combination of JavaScript and PHP is widely used for building web applications.

- A *Virtual Machine* (VM) is a program capable of emulating a - possibly hypothetical - machine. Virtual machines described in this article refer to application virtual machines, which emulate a subset of a machine as a process within the underlying environment. Available environments for implementing this project's virtual machines are of course JavaScript and PHP interpreters.

Obfuscation can only ever try to complicate the analysis of a program and must not be confused with encryption. In fact, it has been shown[5], that obfuscation in the sense of totally eliminating the possibility of reverse engineering is impossible to achieve. And also, by describing a virtual machine designed for obfuscating code, this article opposes its own definition of obfuscation, thus creating a paradox in itself. I will proceed anyway.

The following sections will guide the reader along dazzling design decisions and intimidatingly detailed implementation details to the practical application of two virtual machines executing identical bytecode.

2 Design Considerations

When implementing a virtual machine - or any software project for that matter - the programmer is faced with a number of design decisions. The project's main goal is to come up with an implementation of a virtual machine for use as code obfuscation mechanism. With that in mind, it seems only reasonable to set up the following ground rules as a baseline for design decisions:

1. In order to minimise an attack's payload size, the VM's code size as well as bytecode for the VM should be as small as possible.

2. Implementing the VM for a given bytecode should be fast and simple. Re-using features of the underlying environment - PHP or JS - could be advantageous.[2]

Although some design choices may have been influenced by personal preference, all decisions can relate to these ground rules.

2.1 Feature Set

Should a VM support every imaginable language construct or just the bare minimum? What is the bare minimum? Ground rule 1 suggests to limit features in order to keep the VM small. Ground rule 2 suggests to take advantage of synergy effects by forwarding features to the environment without considerable overhead. Based on these conclusions I decided to support the following features:

- The bare minimum as defined by turing completeness[3]
- Functions
- Variables
- Variable scope by function: Variables defined inside the function body exist only within this function.

[2] The easiest (but worst) way to use the environment would be to simply `eval()` code, which is totally counterproductive in this context and thus excluded.

[3] "To be Turing complete, it is enough to have conditional branching (an *if* and *goto* statement), and the ability to change memory."[8]

- Possibility to call some[4] functions defined in the underlying environment

2.2 Bytecode vs. Direct Execution

Up until now it has been implied that the VM is supposed to process bytecode as opposed to raw source code. The idea of executing source code may seem somewhat farfetched, but there are many examples of programming languages trying to defer source code parsing, e.g. just-in-time (JIT) compilers - such as Java or Python - and also dynamic or context sensitive languages - such as Tcl[5].

However, all of these examples require the source code to be parsed at some point. Since ground rules 1 and 2 suggest a small payload footprint and a simple interpreter implementation, the overhead of a source code parser should not be part of the VM itself. As a result, VMs will be implemented as bytecode interpreters.

2.3 Existing vs. new Bytecode

Re-using existing code, standards and libraries is an essential and very time saving part of every software development process. It stands to reason that the additional effort put into implementing an entirely new bytecode can be self-serving in the context of obfuscation. The whole point of code obfuscation is to disguise code in order to complicate its immediate comprehension. There are plenty of analysis tools readily available for known bytecode implementations, such as Java or .NET bytecode, and none for the newly reinvented wheel bytecode. This certainly qualifies for being complicated.

Even though the task of converting the new bytecode to a well defined equivalent for analysis is not too difficult, it may still be a bit of a hassle for the analysing party.

Since some type of bytecode has to be implemented regardless of being well known in advance, in my opinion the flexibility to change or evolve custom bytecode as suitable slightly outweighs its alternative[6].

2.4 Stack vs. Register Machine

This is a fundamental design decision. Should this virtual machine emulate a stack machine, a register machine or a combination of both? A glimpse into an appropriate publication[6] shows, that a stack machine program has shorter instructions and the overall code size is smaller than the equivalent program for a register machine. This comes at the expense of slightly more operations that have to be implemented with a stack machine for stack management. On the other hand, according to the paper, register machines can have a significant performance benefit over stack machines.

For the purpose of obfuscation - and the reader may disagree here - I decided that a shorter code size is more important than speed. There is the slight overhead of implementing a few stack management operations, however the overall VM

[4] In order to keep it simple, this type of call is limited to global functions without special namespaces or associated classes.

[5] Tool Command Language or better known as Tcl/Tk

[6] During the development phase I stumbled upon UBF(A) and UBF(B)[7], which seemed like a real alternative and may still be considered for future improvements of this project. The Website says "Be warned. Things change frequently", but fortunately the UBF specification remained unchanged since 2002.

implementation of a simple stack machine is still a little easier (and faster) than implementing its pendant.

2.5 CISC vs. RISC

Influenced by *"Yeah. RISC is good."*[1] from the movie *Hackers* I would say that RISC outweighs CISC by far. However this design decision is not solely a matter of personal preference, such as VIM vs. Emacs or Linux vs. OpenBSD. Since it has already been decided to implement a stack machine, there is always the overhead of stack management operations (push, pop, ...) over pure RISC machines. Fast and easy implementation as suggested by ground rule 2 can still be accomplished by being as RISC as possible.

Interestingly, one of a stack machine's traits is a smaller instruction size for each instruction compared to a register machine's instruction size. The term *Reduced Instruction Size Computer* comes to mind, which incidentally matches the cite's acronym *RISC* as well.

2.6 Optimisation

There are three obvious starting points for optimisation: speed, size and obfuscation level. All three points are mutually exclusive to a degree and should be weight according to the context of specific practical applications. E.g. when a scenario comes with a time critical application, optimising the VM for speed rather than size or obfuscation level seems appropriate.

The following ideas come to mind:

- *Speed:* There certainly are more efficient ways to run a program than inside a virtual machine running inside a scripting environment. Apart from optimising the program itself - or letting a compiler perform this task - the VM can try to delegate as much work as possible down to the scripting environment.
- *Size:* Optimising for size can actually mean either dealing with a smaller VM or smaller payload size. In both cases a standard compression algorithm - e.g. zlib - could be applied to solve the problem. In addition the design decision to use a stack machine instead of a register machine has already had a positive effect on minimising the payload size.
- *Obfuscation level:* Enhancing obfuscation is a task only limited by the programmer's creativity. Just a few examples: The stack machine could implement multiple stacks and unexpectedly switch between them. The meaning of each op-code could change regularly, e.g. after each op execution. There could be yet another VM running inside the VM. Junk code or junk data could try to mislead the unskilled analyst.

3 Implementation Details

This chapter briefly describes specifics about the implemented bytecode, the compiler - yes, there is a compiler - and further thoughts about the implementation.

3.1 Bytecode Encoding

All instructions start with a single byte denoting the op-code, optionally followed by data specific to the operation. A complete list of op-codes can be found in the project[2] documentation.

3.1.1 Datatypes

In order to be turing complete it is not absolutely necessary to implement any datatype at all. Data could simply be entered into the program as instructions only - e.g. "+++" in *Brainfuck* increments a register three times thus expressing the number three and the ASCII character coded by the number three respectively. However in favour of program code size and performance two of the most used datatypes have been implemented - integers and strings. They are being encoded like so:

- Integer

  $\|$ ID (0x69) $\|$ sign bit s (1 bit) $|$ length L (7 bits) $\|$ L bytes, big endian $\|$

  Example: $\underbrace{\texttt{0x69}}_{\text{ID}}$ $\underbrace{\texttt{0x02}}_{L=2}$ 0x05 0x39 $\iff 1337$

- String

  $\|$ ID (0x73) $\|$ string length L encoded as Integer $\|$ L bytes $\|$

  Example: $\underbrace{\texttt{0x73}}_{\text{ID}}$ $\underbrace{\texttt{0x01 0x04}}_{L}$ $\underbrace{\texttt{0x74 0x65 0x73 0x74}}_{\text{"test"}} \iff$ "test"

Other datatypes such as arrays, objects or hash-tables can always be emulated by calling interface functions from the underlying environment.

3.1.2 Other Instructions

- Functions: The *echo* function is defined as bytecode operation. Other functions can be defined and called.
- Stack operations: POP and SWAP have been defined. A PUSH operation is implicit for all data (integers/strings) found in the bytecode and for operation results.
- Branches/Conditions: One bytecode has been defined each for unconditional jump, conditional jump and negated conditional jump.
- Variables: Variables can be assigned or looked up. Each function has its own variable scope. Global variables from the underlying environment can be accessed.
- Expressions: The usual mathematical operators (plus, minus, modulo, ...), binary operators (and, or, xor, shift), logical operators (and, or, xor) and comparison operators (equals, greater, smaller, ...) are supported.

3.2 Language Quirks and Compatibility

One specific detail about implementing a bytecode parser is whether the bytecode lookup should be done by hashtable (or equivalents) or by switch statement. A hashtable lookup brings a lot of flexibility into the VM for runtime changes or later improvements, whereas a switch statement is trivial to implement and potentially faster (which is a wild assumption). The current implementation is using a switch statement, but this may change in the future.

Another question is how to call the VM from the environment - flat functions (like init(); run("code");) or object oriented (e.g. a = new VM(); a.run("code");). For compatibility with older versions of PHP (<5) the PHP VM is implemented as flat function layout. Since every major implementation of JavaScript (e.g. v8 or seamonkey) is OO-aware, the JS version of the VM is OO.

The list of subtle implementation details goes on forever. The interested reader is welcome to read the source code.

3.3 Compiler

Writing pure bytecode all the time turned out to be rather tiring. In order to ease software development for the new VMs a simple compiler has been added to the project's repository[2]. Its source language is a subset of PHP called EPHP - output is the bytecode in question, optionally encoded as base64 or JSON string. The compiler is written in Erlang, which makes it straight forward to read and understand its inner workings.

Any compiler optimisations have been omitted and left as an exercise for the reader.

4 Practical Applications

4.1 Preparation

4.1.1 Starting the VM

Using the VM in either JavaScript or PHP can be easily shown by example. The following code snippet embeds a simple example bytecode in HTML/JavaScript:

```
<html><body>
    <script src="../vm/vm.js"></script>
    <script>
        __vm_run("s\x01\x04test\x01");
    </script></pre>
</body></html>
```

The same example can be run with this standalone PHP script:

```
<?php
include("../vm/vm.php");
__vm_run(base64_decode("cwEEdGVzdAE="));
```

Or it can be called from command line:

```
php vm.php cwEEdGVzdAE=
```

The bytecode example used in these examples was generated by the bytecode compiler from this hello-world EPHP script:

```
echo "test";
```

The following two command lines generate JS or base64 encoded bytecode respectively:

```
./scripts/ephpc.erl -js samples/ephp/hello_world.ephp --
./scripts/ephpc.erl -base64 samples/ephp/hello_world.ephp --
```

4.1.2 I/O

Now, in order to properly use the virtual machines there must be well defined inputs and outputs.

- Input/Output: access external global variable scope

  This simple concatenation example makes use of the feature, that externally defined variables - in this case *$in* and *$out* - can be read and written.

  EPHP code:

  ```
  $out = "test" . $in;
  ```

  PHP code:

  ```
  <?php
  include("../vm/vm.php");
  $in = "abc";
  $out = "";
  __vm_run(base64_decode("cwEEdGVzdHMBAmluZy5zAQNvdXRhUA=="));
  echo $out;
  ```

- Input/Output: access external functions

  Input and output data can be channelled through function calls like so:

  EPHP code:

  ```
  foo("test". bar());
  ```

  PHP code:

  ```
  <?php
  include("../vm/vm.php");
  function bar() { return "123"; }
  function foo($input) { echo $input; }
  __vm_run(
    base64_decode("cwEDZm9vcwEEdGVzdHMBA2JhcmkAZi5pAQFmUA==")
  );
  ```

- Input/Output: predifined variables

  The object oriented implementation approach of the JavaScript version of the VM allows to inject variables into the VM variable scope prior to running code.

  EPHP code:

  ```
  $out = "test" . $in;
  ```

  HTML/JS code:

  ```
  <html><body>
      <script src="../vm/vm.js"></script>
      <script>
          vm = new __vm();
          vm.v.set('in', 'abc');
          vm.run("s\x01\x04tests\x01\x02ing.s\x01\x03outaP");
          document.write(vm.v.get('out'));
      </script></pre>
  </body></html>
  ```

- Output: echo

  Of course the easiest output method is to simply print results. In case of JavaScript running within a web browser printed results can later be read by DOM access.

4.1.3 Other preliminary considerations

A real life scenario presents various obstacles trying to prevent us from executing suspicious code. Even though strings like *cwEEdGVzdAE=* may not look suspicious to the naked eye, but web application firewalls and input filters may decide otherwise. Naturally, encoding payload appropriately to elude such filters should be given some thought.

In addition it can be useful to encrypt payload or VM or apply more layers of obfuscation. A sample chain of events could look like this:

- encrypt VM -> encode VM -> transfer VM -> decode VM -> decrypt VM
- compile payload -> encrypt payload -> embed decryption routine in payload -> encode payload -> transfer payload -> run payload in VM

4.2 Hide Code and Data

A rudimentary application for this project is to hide code and data. The following example shows a simple XOR based encryption routine (EPHP code) hidden inside *$code* in the PHP part. *$data* holds encrypted data which was encrypted using *$code* as key.

EPHP code:

```
function scramble($key, $data)
{
    $out = "";
    for ($i = 0; $i < strlen($data); ++$i) {
        $keychr = substr($key, ($i % strlen($key)), 1);
        $datachr = substr($data, $i, 1);
        $out .= chr(ord($keychr) ^ ord($datachr));
    }
    return $out;
}

$data_out = scramble($key, $data);
```

PHP code:

```
<?php
include("../vm/vm.php");
$code = base64_decode( "cwEIc2NyYW1ibGVzAQNrZXlzAQ...RhUA==");
$key = $code;
$data = base64_decode("P256Fg5SCB0RGQhTZWwHCgt...QobXA==");
$data_out = "";
__vm_run($code);
echo $data_out;
```

Both base64 encoded values in *$code* and *$data* are abbreviated for clarity.

4.3 Metamorphic Code

Slightly simplified, code that can change itself is called metamorphic code. The following example shows how to execute *$code* over and over while the code itself (EPHP code) is able to access its own programming until the modified program is being executed in the next iteration cycle. The self-modification is not too sophisticated, but the idea becomes apparent.

PHP code:

```
<?php
include("../vm/vm.php");
$code = base64_decode("cwEGc3RybGVucwEEY29kZWdp...cnVuYVA=");
$run = true;
while ($run)
    __vm_run($code);
```

EPHP code:

```
if (strlen($code) > 2000) {
    echo "A";
    $run = 0;
} else {
    echo "B";
    $code .= $code;
}
```

A suitable extension to this idea would be to generate PHP code inside the VM, which in turn generates *$code*.

5 Conclusion

At the end of this article I would like to recapitulate and question everything. First of all: Has the obfuscation been successful? Yes. No. There are plenty of attack vectors against obfuscation which can never be eliminated. But as far as reverse engineering is concerned, it is certainly harder to analyse a bytecode program now than it was to read PHP or JavaScript source code before the obfuscation process.

Were all the design decisions correct in retrospect? A stack machine, limited feature set and the potential for optimisation turned out to be a suitable combination. During the research phase of this project I bounced a few ideas off some friends. Curiously everybody who thought about virtual machines in detail before, favoured register machines. I wonder why.

I would consider this project to be complete for the moment, but when has a software ever been completed? There is plenty of room for enhancements: Due to a lightweight feature set it would be easy to support more host languages besides PHP and JS. And there is need for a standard library to improve compatibility between different VM implementations. The todo- or nice-to-have-list goes on forever.

Finally, and even though obfuscation is only security by obscurity, it has been lots of fun exploring ideas about bytecode interpreters and abstraction layers, virtual machines and compilers. I would be delighted to find a program using this VM concept out in the wild one day.

References

[1] Hackers (movie) transcription. http://www.movietranscriptions.com/209177_Hackers_1995.html#p540, 1995. [Online; accessed 24-March-2010].

[2] Ephp compiler and bytecode interpreter in php and javascript. http://code.google.com/p/ephp-vm/, May 2010.

[3] Programming language popularity. http://www.langpop.com/, March 2010. [Online; accessed 30-March-2010].

[4] Tiobe programming community index for march 2010. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, March 2010.

[5] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. http://www.cs.princeton.edu/~boaz/Papers/obfuscate.ps, November 2001. [Online; accessed 25-March-2010].

[6] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: Stack versus registers. http://www.usenix.org/events/vee05/full_papers/p153-yunhe.pdf, June 2005. [Online; accessed 29-March-2010].

[7] unknown. Ubf home. http://www.sics.se/~joe/ubf/site/home.html, March 2002. [Online; accessed 27-May-2010].

[8] Wikipedia. Turing completeness — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Turing_

completeness&oldid=346135365, February 2010. [Online; accessed 09-April-2010].

Acronyms

JS JavaScript

PHP PHP Hypertext Preprocessor

VM Virtual Machine

BC Bytecode

CISC Complex Instruction Set Computer

RISC Reduced Instruction Set Computer