Winning the Online Banking War

Sean Park

Senior Malware Scientist

TrendMicro

BLACKHAT USA 2015

Overview

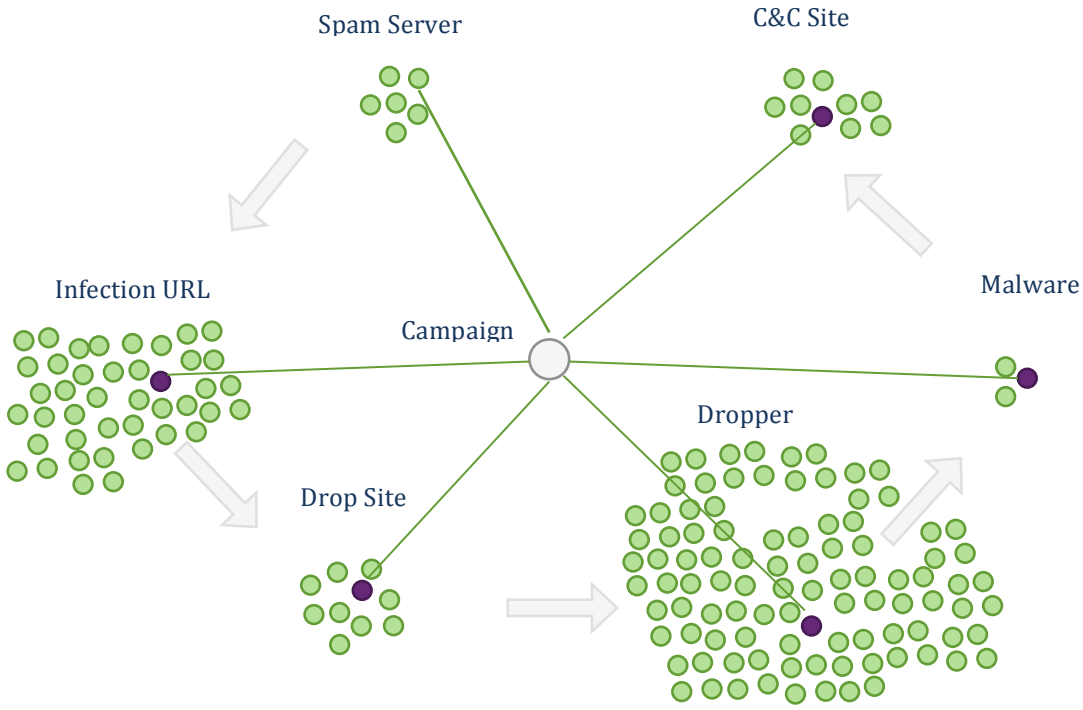
Cyber-attack has been increasingly prevalent over benign internet community ever since the dawn of internet. Malware plays a crucial role in this, enabling automated execution of cyber-attack. One of the major goals of these cyber-attacks is obviously, 'stealing the money', which introduced a special category of 'Banking Trojan' (or Banking Malware). The advent of 'Zeus' ignited the renaissance of banking Trojans a few years ago. The Zeus source leak enabled the underground community to competitively develop new variants and even pushed the creation of fresh banking Trojan families such as Carberp.

This boom of banking Trojans was possible because Zeus model allowed a modular approach that separates malware from money-stealing web application logic - which is called the 'Injection'. This injection enables cybercriminals to steal online banking customer credentials and to perform transaction manipulation and injection while bypassing two-factor authentication. Although detecting banking malware binary and evading the detection is significant part of security industry, the detection of inject and the evasion became a new hot battle ground between banking industry and cybercriminals. This paradigm forces the banking industry to work on a new protection framework.

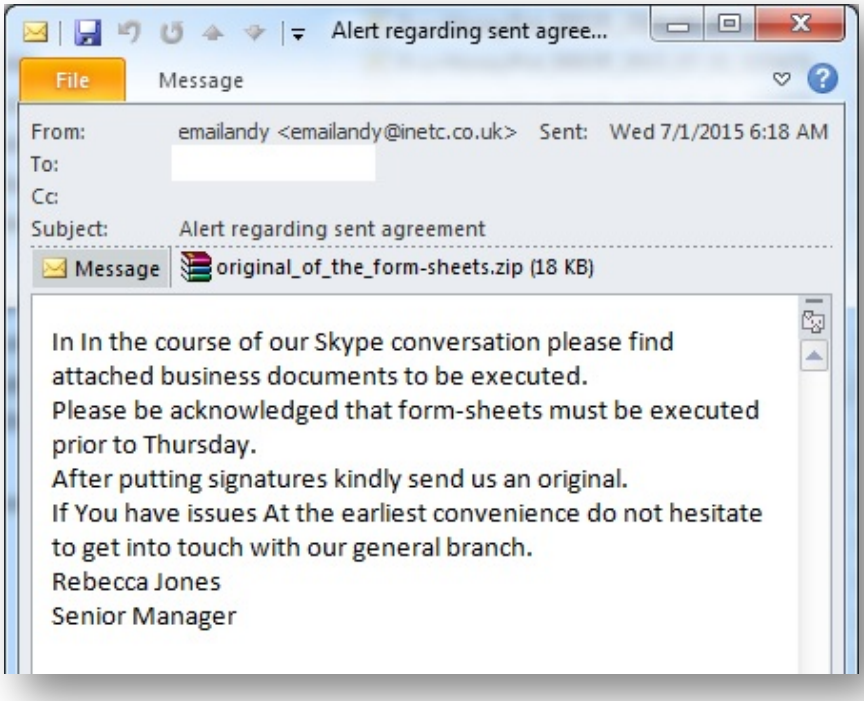
This white paper is an attempt to identify the crux of online banking war and set the strategic defense framework for the banking industry.

Banking Malware Campaigns

The majority of cyber-attacks consist of a complex life cycle, which includes exploitation of software vulnerabilities, infiltration into the victim's machine, and exfiltration of information. Modern cyber-attack campaigns require many supporting infrastructure including spam servers, social engineering engine, metamorphic malware, and so on. Banking malware campaigns require additional application logic, Inject, which is a JavaScript that gets injected into the browser and compromises online banking sessions. A general banking malware campaign is illustrated below.



One of the major banking malware delivery mechanisms is socially engineered email. See an example below.



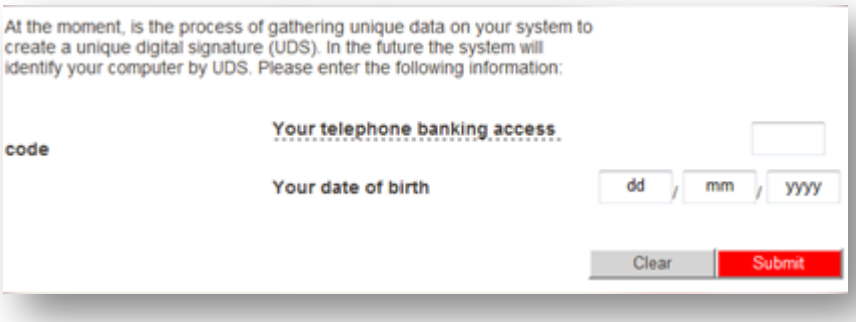
This social engineering scheme may not come quite convincing to many of you. That's why the attackers use many different types of social engineering schemes.

Attack Mode

Once the victim's machine is infected, the banking malware now starts the core application logic by monitoring the online banking pages the user accesses and injecting malicious scripts into them. There are several attack modes in stealing victim's fund from the online banking account. Some of the key MOs (Modus Operandi) are described below.

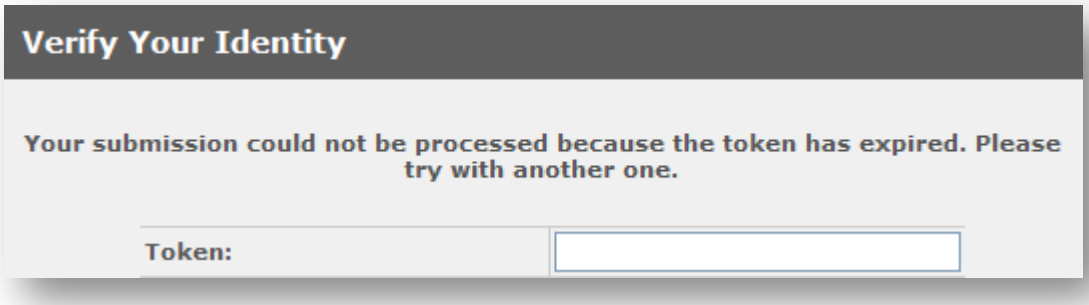
Sign In – Phone Banking Code

The following inject asks the online banking user to enter extra user credentials required to steal the money through phone banking. The psychological aspect of this attack is that users trust anything displayed when they enter user name and password correctly.

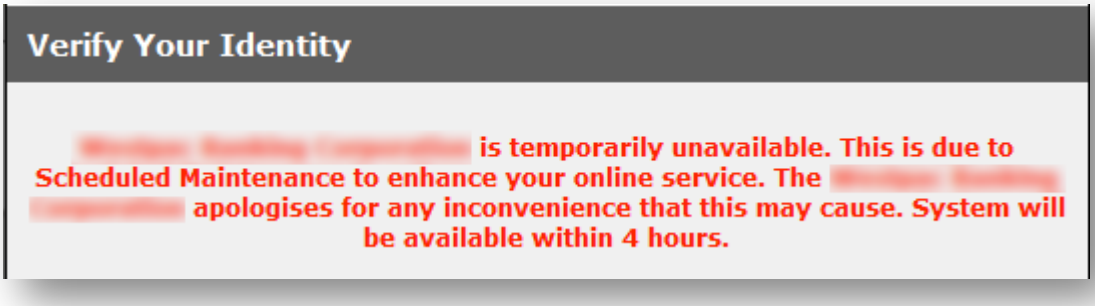


Sign In – Token

Many banks deploy non-transaction bound authorization method using one time passwords such as tokens. Again the online banking customer will blindly trust the web UI rendered by the inject once they are logged in.

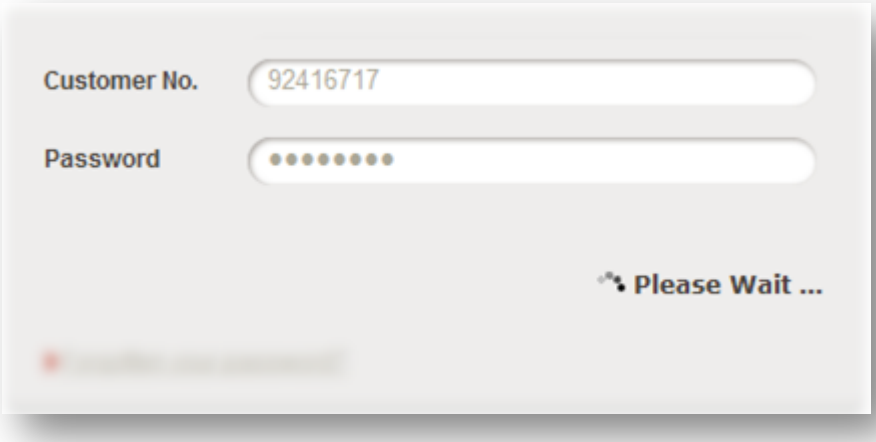


In this case, the cybercriminals go smart on buying some time to make transactions before the online banking customer sounds an alarm to the bank

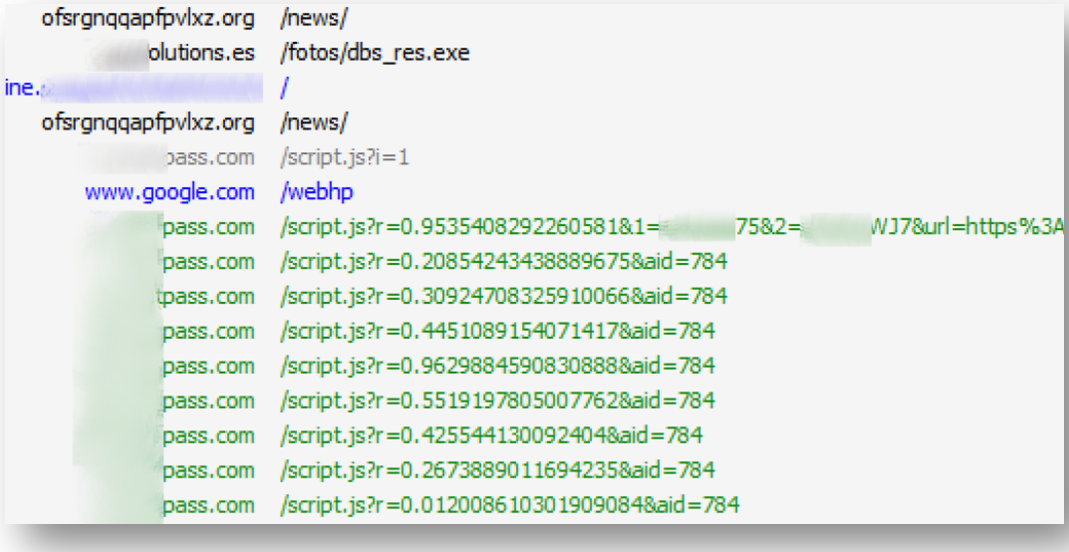


Sign In - MITM

When the online banking customer logs in, 'Please wait' message appears for unusually long period before being allowed to the banking page.

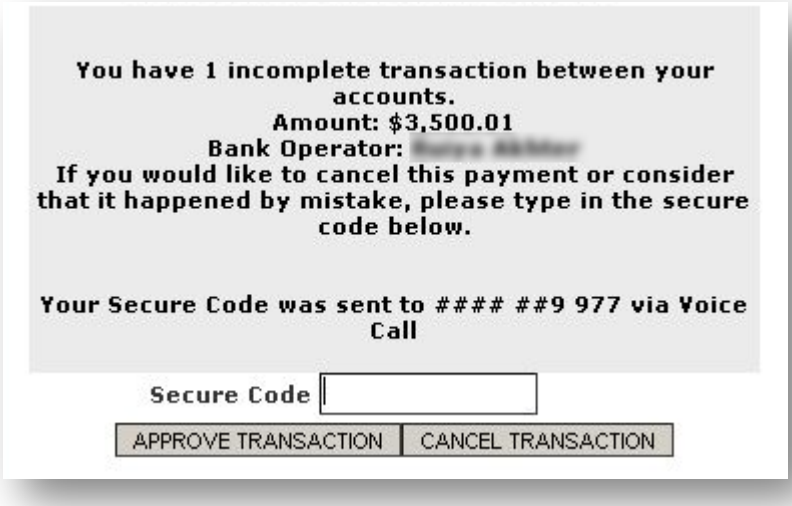


Looking at the network traffic log, the victim's machine simply polls the C&C waiting on a response back. This turned out that the crooks perform transactions while proxying through the victim's machine within the authenticated banking session.



Transaction Injection

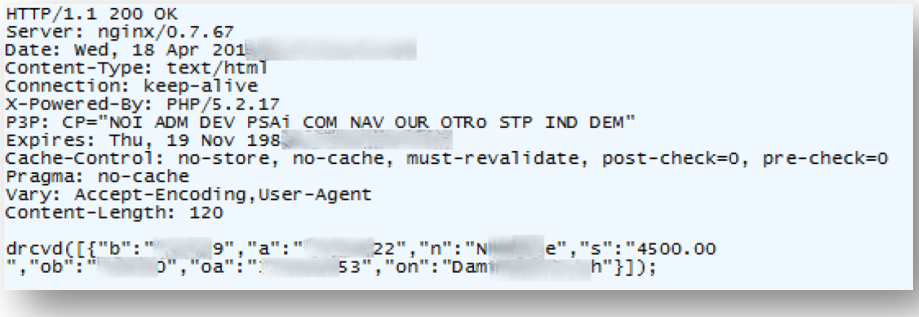
The following inject gets triggered within an online banking session. Since many banks deployed two factor authentication methods such as transaction bound mobile phones and tokens, any unexpected transaction notification will immediately sound an alarm to the online banking customers. This inject uses a social engineering technique when injecting a transaction.



Transaction Manipulation

The following picture shows the packet log when the customer makes a transaction. The highlighted packets show that the inject dynamically retrieves a money mule's account details from the C&C and manipulates the transaction the customer made.

The following packet capture shows dynamic money-mule information retrieval at run time.



MIPS

MIPS (Malware Inject Prevention System) is a hypothetical online banking security framework that we would like to create to defend against banking malware. People might ask why we need MIPS as we have anti-malware solutions. The reality is it is difficult for many banks to offer anti-malware solutions to every online banking customer. In addition, as is quite often the case, a small window between malware infection and malware pattern update is good enough for malicious transactions even if the customer's machine is equipped with an anti-malware solution. Instead of trying to discover malware injection in browser's virtual memory space, MIPS attempts to find out the JavaScript and associated artefacts injected by the malware. In terms of that, MIPS works in web application level instead of operating system level.

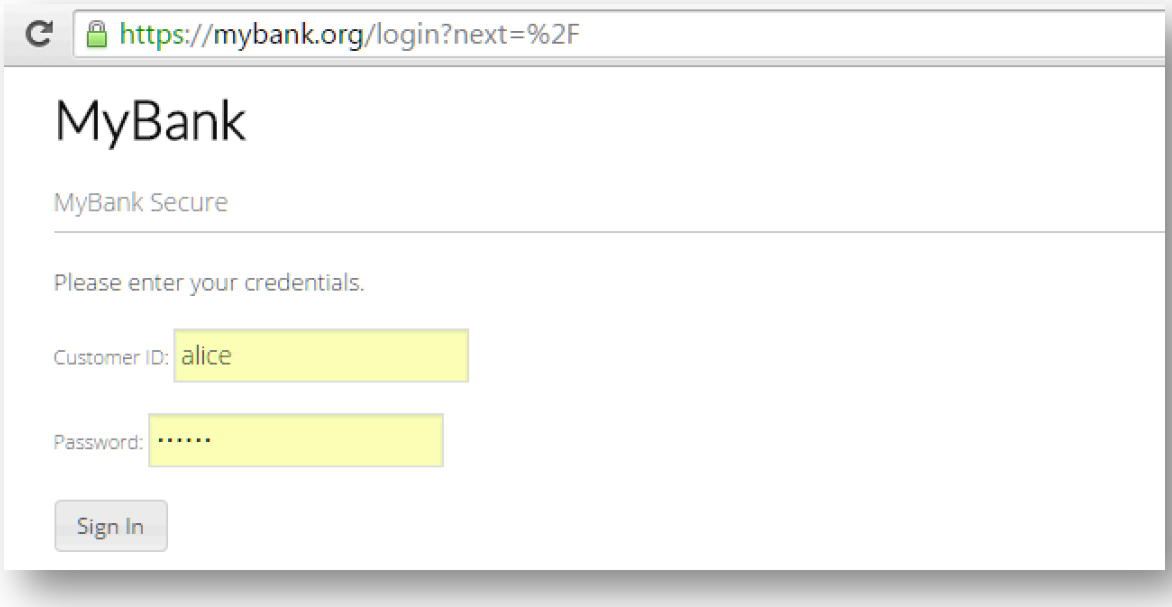
Note that this white paper uses tamper monkey to demonstrate the activity of banking malware's JavaScript injection into web pages. Tamper Monkey and Grease Monkey are popular browser plug-ins that allows custom JavaScript injection. Although tamper monkey successfully simulates dynamical script injection behavior, keep in mind that banking malware can also statically replace MIPS code before a single line of MIPS code is executed.

It should be highlighted that using HTTP over SSL tunnel does not help security at all in the presence of banking malware since HTTPS is designed for securing the data on the wire whereas banking malware takes full control over the machine intercepting decrypted HTTP packets in the browser and injecting malicious scripts within the HTML pages.

Now that we know the banking malware threat landscape, let's find out how the real online banking war goes between the cybercriminals and the defenders.

DOM Injection

In this white paper, the attack and defense will be demonstrated using a hypothetical online banking page, https://mybank.org. The sign-in page is shown below.



Attack: DOM Injection

The simplest form of DOM injection is to install 'click' handler on the 'Sign In' button at the sign-in page. The following inject demonstrates a jQuery based implementation of click handler that steals the customer ID and the password.

```
$("#submit").on("click", function(){
    var id = $("#signin-id").val();
    var pw = $("#signin-password").val();
    console.log(">> DOM Inject: "+id+:"+pw);
});
```

Note that this inject can be installed either on page load or directly embedded within the actual online banking page before it gets rendered by the browser.

Defense: DOM Scan

In order to detect the malware inject, MIPS can simply scan the runtime DOM (Document Object Model) and find malicious JavaScript. There are many variations in defining the scan data for detection. Common scan data include:

- Script hashes (i.e. MD5, SHA1, TLSH)

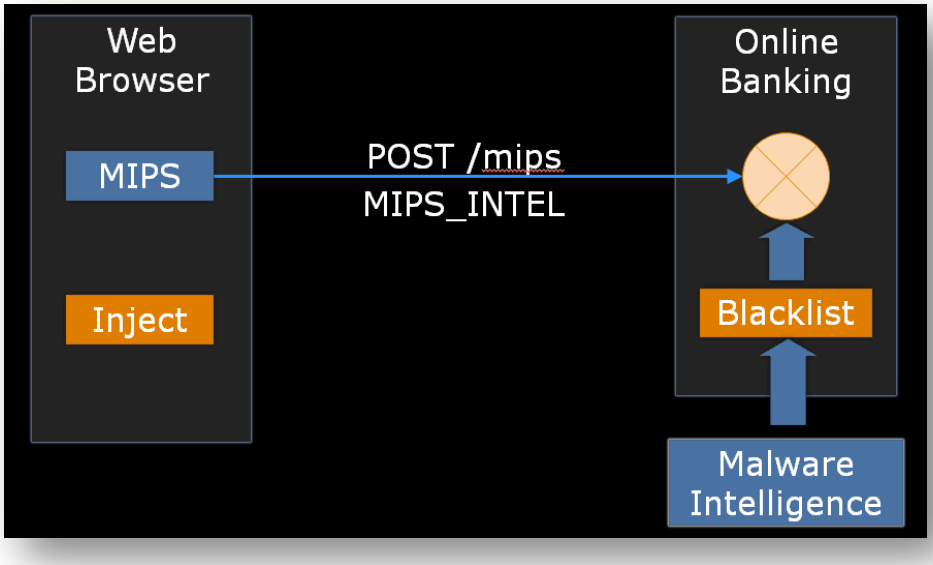
- Zipped raw script contents
- DOM hierarchy and statistics

Each one of these approaches their pros and cons and is also potentially subject to EULA of online banking. Throughout this white paper, a script hash scan method is used for demonstration purpose.

MIPS requires at least the following three components.

- MIPS JavaScript
  - It runs on the online banking page that needs to be protected
  - It scans the DOM and sends the result back to the online banking server
- Malware Intelligence
  - Malware inject signature or pattern needs to be collected through various channels such as in-house malware analysts, inter-bank data exchange, or third party vendor.
  - Blacklist keeps these records
- MIPS Fraud Analytics System
  - It saves the transmitted MIPS log in the database
  - It performs the infection check by correlating blacklist and MIPS log.

MIPS baseline implementation is depicted below.



This is essentially the method that many vendors and banks are currently using to detect the presence of banking malware.

A straightforward DOM script scan implementation can be found in the Appendix A. Please be aware that third party browser plugins can introduce scripts. Blindly flagging all unknown JavaScripts will cause a massive False Positives. It is important to maintain malware intelligence for accurate detection.

DOM Stealth

Attack: How It Works

In response to naïve DOM scan based detection, malware can choose to compromise the integrity of the DOM scan data MIPS retrieves. As long as script enumeration result proves to be clean, no infection will be detected.

When the malware inject removes its own ***script*** node from the DOM while its reference is kept in other part of the DOM, it this causes a memory leak since JavaScript's garbage collector won't be able to free the memory object whose reference count is positive. As a consequence of this, malware inject's function will be activated when the event handler it registered gets triggered. More importantly the naive script enumeration technique will not be able to see the removed ***script*** node.

There are several memory leak patterns exploitable by the malware inject to achieve DOM stealth including:

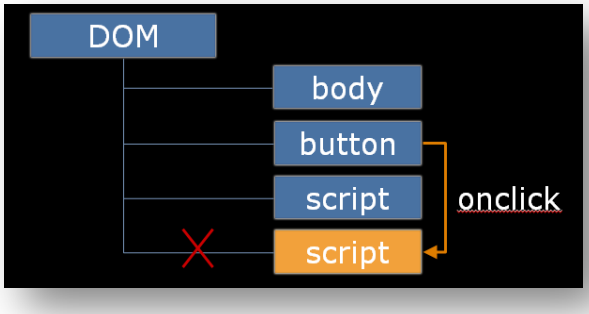
- Dangling references
- Circular references
- Closures

Dangling References

A typical example is shown below.

```
01 $("#submit").on("click", function(){
02     var id = $("#signin-id").val();
03     var pw = $("#signin-password").val();
04     Steal(id,pw);
05 });
06
07 function Steal(id,pw){
08     console.log(">> DOM Stealth: " + id + " / " + pw + "!");
09 }
10
11 // Now delete this script
12 // jQuery way: $("#inject").remove();
13 var me = document.currentScript;
14 me.parentNode.removeChild(me);
```

Line 1 installs click event handler on 'Sign In' button, which creates a reference to the function defined in line 1 through to 5. Line 12 shows jQuery way of removing the above script node whereas line 13-14 demonstrates a standard JavaScript DOM node removal. If the above script runs, a naïve DOM scan algorithm will not detect the above inject. A conceptual diagram of this technique is illustrated below.



Circular References
An example of using circular reference is shown below.

```
01 var refToDom = document.body;
02 document.body["refToScript"] = refToDom;
```

As you can see this extra step of adding a reference to a DOM object within the malware inject is not really necessary. Its included here for completeness's sake.

Closures
The following code snippet shows a closure version of the implementation.

```
01 function AttachEvent(element) {
02  element.attachEvent("onclick", Steal);
03
04  function Steal() {
05      /* This closure references element*/
06      var id = $("#signin-id").val();
07      var pw = $("#signin-password").val();
08      console.log(">> DOM Stealth: " + id + " / " + pw + "!");
09  }
10 }
11
12 AttachEvent(getElementById("submit"));
13
14 var me = document.currentScript;
15 me.parentNode.removeChild(me);
```

The closure inside AttachEvent() function will remain in the memory even after the above script is removed from DOM.

Defense: DOM Event Scan

The task of scanning the entire physical memory in an attempt to find the hidden script element seems to be quite computationally expensive as well as error-prone as the scan needs to address various issues arising from different browser types let alone different browser versions. If you focus on the fact that malware injects need an entry point function defined somewhere in DOM space, a more efficient and feasible approach would be:

- Identify all possible entry points of malware inject (i.e. unload, click, timer, etc)

- Enumerate all event handlers

In JavaScript there are many ways to register an event handler. Most notable methods and how they can be enumerated are summarised below.

Event Registration	Event Discovery
element.onclick = handler	element.onclick
element.addEventListener	getEventListeners(element, “click”)
$(element).on(“click”, handler)	$._data(element, "events")
$(element).observe(“click”, handler)	element.getStorage().get('prototype_event_registry').get('click')

Note the above table was derived from the following sources:
http://stackoverflow.com/questions/446892/how-to-find-event-listeners-on-a-dom-node/447106#447106
http://stackoverflow.com/questions/2518421/jquery-find-events-handlers-registered-with-an-object

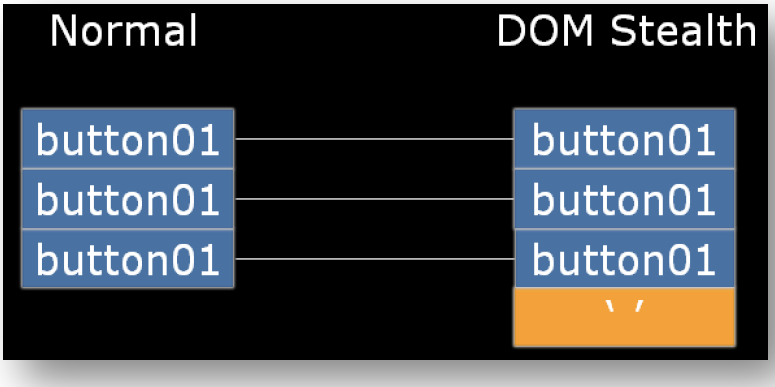
An example event handler enumeration using jQuery ***_data()*** method is shown below.

```
01  var clickEvents = $._data(document.getElementById("submit"), "events").click;
02  $.each(clickEvents, function(key, obj) {
03      console.log(obj.handler)
04  });
```

Note that there could be non-standard enumeration implementations across different browsers and versions.

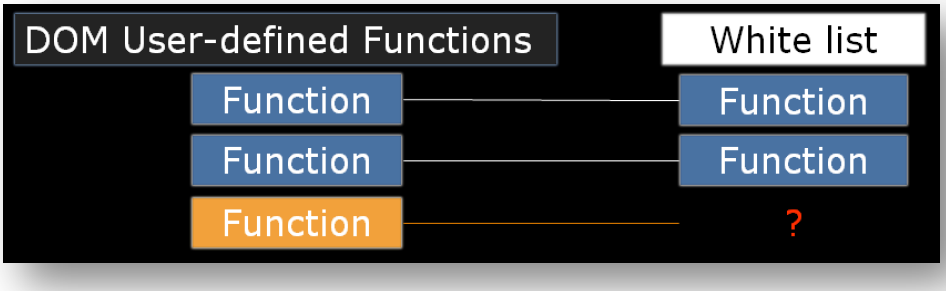
Defense: Artefact Analysis

Once all event handlers in question are identified, we need to find whether the event handler function resides within a removed script node. There is no standard way of achieving this goal. One of the possible methods to discover hidden DOM script is to perform artefact analysis. For instance, all event handlers registered in normal way that remain connected to DOM have its ***namespace*** property set to a valid value (i.e. ‘button01’, ‘button02’) whereas those handlers pointing to hidden DOM element all have an empty string. This artefact analysis is one of the methodologies that can be deployed for stealthy DOM object. In the following diagram, namespace value of a clean web session is shown on the left while the button click event handler installed through DOM stealth method has an empty namespace string (marked in orange) on the right.



Defense: Function Integrity Check

A more robust approach than artefact analysis to discover hidden DOM script is to use function integrity check. Everything in JavaScript is implemented as an *object*. All currently active variables, functions, and objects can be enumerated using ***Object.keys()*** function. The detection strategy is simple. MIPS compares all enumerated JavaScript objects with pre-computed whitelist. See the conceptual diagram below.



The steps are as follows:

- Enumerate user-defined functions
    - *Object.keys(window).filter(!/\[native code\]/)*
- Compare functions discovered in DOM against whitelist

An example user-defined function enumeration source code can be found in Appendix B.

Although the strategy is simple enough, the devil is in the detail. There are a couple of implementation challenges. First of all, where would MIPS check the integrity? You can send out whitelist to the client and let the check be performed in the browser on the fly. Alternatively you can keep the whitelist on the server side and let MIPS script just send full object enumeration data back to server for a check. In general, it's more secure to keep your detection decision logic obscure to the attacker. Therefore it's recommended to do the check on the server side.

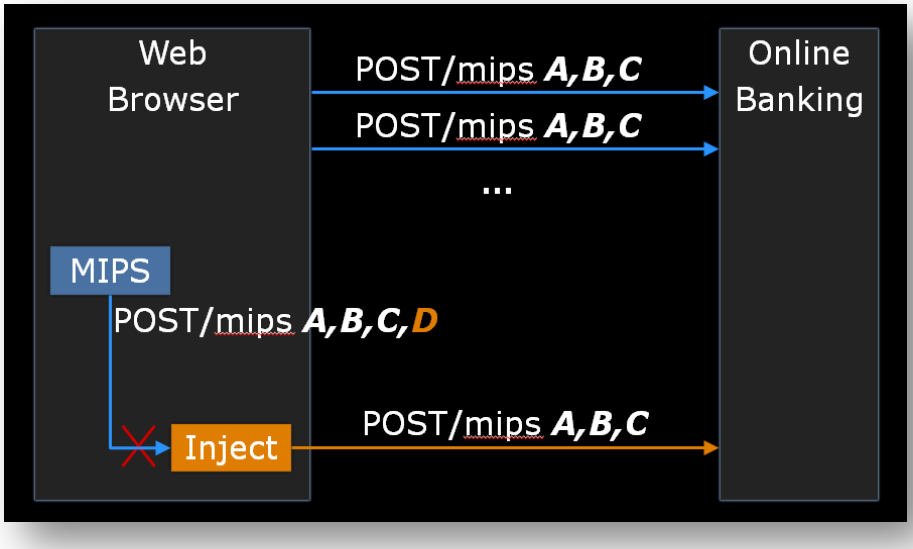
Another challenge is the size of the object enumeration data. There are at least a couple of hundreds of function objects. You can zip the source, or send hashes of functions. There are several alternative approaches on this problem.

MIPS Infiltration

Obviously the attackers can infiltrate into the core of MIPS, eavesdropping on MIPS communications, intercepting, modifying, and forging MIPS code and data since the malware inject runs in the same operating environment with superior privilege.

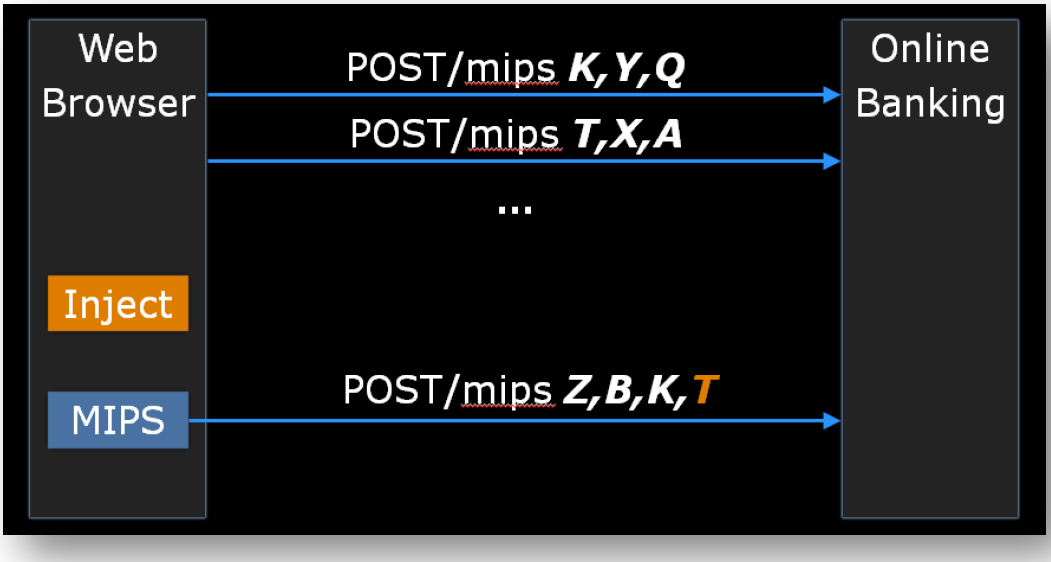
Attack: Replay

Traditional replay attack can be performed on MIPS traffic. Once attackers identify the pattern of AJAX POST requests initiated by MIPS script, they can simply block the MIPS POST requests by hooking the AJAX call or directly manipulating MIPS code, and send a 'clean' replay of the AJAX POST request. The following diagram illustrates this attack where the attacker monitors AJAX POST requests sent by MIPS script that have hash values of ***A***, ***B*** and ***C*** in an uninfected browser, and replays it after deactivating MIPS script's POST request that contains additional hash value of ***D***.



Defense: Salting

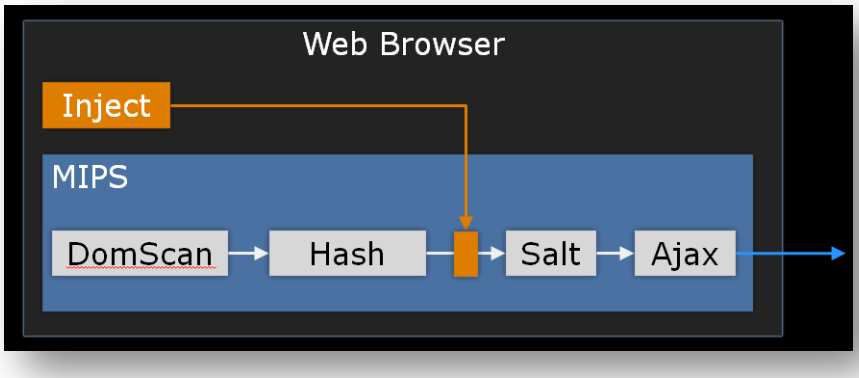
A well-known defense for replay attack is to use salting (a.k.a. session tokens). In order to defeat replay attack, the online banking server returns a random number, salt, for each HTTPS GET request to a page, which is used by MIPS script that transforms the collected scrip hashes into scrambled values, which then get reversed by the online banking server to get the original hashes. Since the malware observes changing hashes over time while running as a man in the middle, it is unable to perform replay attack in the presence of salting on MIPS intelligence being transmitted on the wire. This salting technique is illustrated in the following diagram where MIPS script returns different hashes for different sessions.



Attack: Forging MIPS Intelligence

One of the counter-attacks on salting is to directly interfere with MIPS execution by intercepting MIPS functions and forging the data conveyed between internal MIPS functions. MIPS is merely a JavaScript. Malware can intercept MIPS script's control flow statically by modifying MIPS script before it executes, or dynamically by implementing a proxy pattern on a MIPS function. Proxy pattern is essentially DOM function hooking.

The following diagram depicts this forging attack where malware inject inserts its code in between ***Hash()*** and ***Salt()*** MIPS functions. Malware can simply delete the hash value corresponding to malware inject from ***Hash()***'s result and pass it on to ***Salt()***.



Attack: Model 1

Let's take a look at the generalised attack model for MIPS intelligence forgery. The following code snippet demonstrates a static code insertion on the original MIPS code.

```
01 var scripts = DomScan(mips_code);
02 scripts = Modify(scripts);
03 var hashes = Hash(scripts);
04 var salted_hashes = Salt(hashes);
05 var check = CheckIntegrity();
06 check = Modify(check)
07 Ajax(mips, salted_hashes, check);
```

Malware-inserted lines are highlighted in red. Line 2 inserted by malware modifies the results returned from ***DomScan()*** MIPS function while line 6 invalidates the MIPS integrity check. In general, malware can insert or replace MIPS code with bypass code through the following techniques.

- Regex (after de-obfuscation if there is any)
- On-the-fly control flow analysis
- Insert a callback function and modify the result

In addition, malware can tamper with intermediate MIPS data being conveyed in the pipeline within the local function/closure or the data DOM integrity check computes

Attack: Model 2

While attack model 1 interferes with MIPS control and data flow, the attacker can also reverse engineer MIPS, deactivate MIPS code and simulate MIPS code with bypass code. See an example below.

```
01 // Deactivate MIPS code
02 //var scripts = DomScan(mips_code);
03 //var hashes = Hash(scripts);
04 //var salted_hashes = Salt(hashes);
05 //var check = CheckIntegrity();
06 //Ajax(mips, salted_hashes, check);
07
08 Var hashes = clean_hash_set;
09 var salted_hashes = Salt(hashes);
10 Ajax(mips, salted_hashes, clean_integrity_check);
```

In this model, using the pre-computed clean data in line 8 and 10, malware can successfully bypass MIPS by invoking necessary MIPS functions.

Defense: Bad Defenses

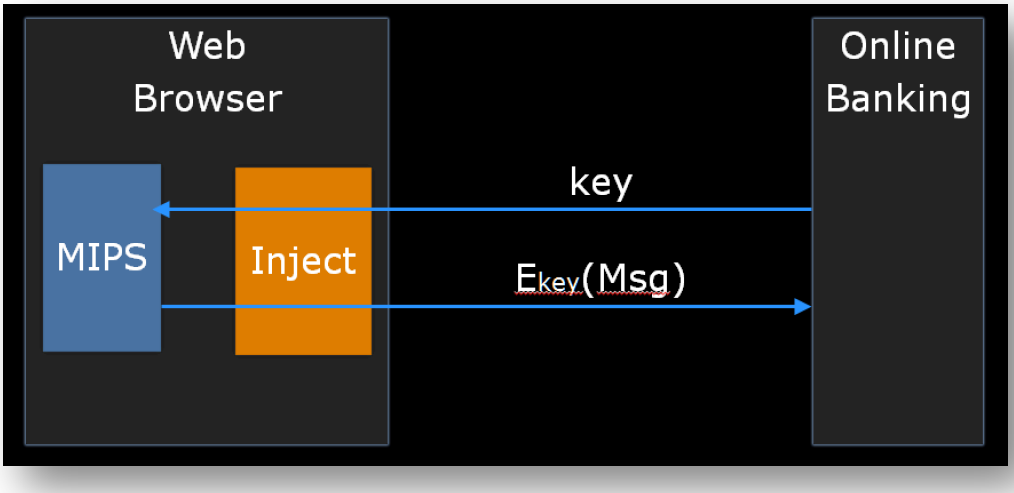
Attacker Workflow

Before discussing defense model, it is important to understand the attacker's reverse engineering workflow. Some of the obvious methods include the following:

- Dynamic analysis
  - Use static analysis tools (Google closure compiler, spider monkey, custom tools, etc)
  - Understand program structure by setting breakpoints and evaluating expressions
  - Bypass dead code
  - Monitor network traffic
  - Targeted reverse engineering by searching keywords (i.e. 'script', '/mips')
- Activity monitoring
  - Hook key functions (i.e. getElementsByTagName, $.ajax, etc)

Obfuscation
With this in mind, obfuscation is not resistant to targeted code inspection/modification as it is vulnerable to dynamic analysis.

Encryption
Another incorrect approach is to blindly use an arbitrary encryption method for data being transmitted between the client and the server, or for cross-function communications. No matter how secure the deployed shared key crypto algorithm is, the key and the crypto code are visible to the attacker. From the attacker's point of view, if the input and the output of a complex function block are predictable, there is no need to fully dissect the crypto algorithm because the attacker can simply call those functions using the key available to the attacker. In short, encryption based defense approach is vulnerable to dynamic analysis



More
Likewise, blindly applying well-known traditional metamorphic/polymorphic techniques without understanding the attack vector will fail. Some of the common methods and their pitfalls are explained below.

- Dead code insertion
  - Attackers can focus on interesting code only (i.e. 'script' selector, salting/hashing). Regex also works in the presence of dead code.
- Name Polymorphism
  - Simple variable/function name polymorphism can be defeated by hooking and dynamic analysis if code structure remains the same.
- Control flow manipulation
  - This method is vulnerable to regex and hooking.
- Function restructuring
  - This method is vulnerable to regex and hooking.
- Opaque predicate
  - This method is vulnerable to dynamic analysis.

Defense: MIPS Code Integrity Check

One of the key strategies to defeat MIPS intelligence forgery is to ensure the integrity of MIPS code. The attack model has a limitation where bypass code must be ***inserted*** in between MIPS code. Defenders can wrap MIPS integrity verification logic around MIPS main logic by exploiting malware's limitation.

The following code snippet demonstrates one of *many* possible code integrity check algorithms. Nested nature of JavaScript closures enables the closure-based nested call context retrieval to reveal inserted code. In each function, ***arguments.callee*** returns the current function source while ***arugments.callee.caller*** returns the caller function source. Since this example takes advantage of nested feature of closure, ***arguments.callee.caller*** on outermost function will return all nested closures. Any attempt to insert code in this structure will change the final return value that is computed using the forward path nonce, na, and return path nonce, nb that are random numbers generated in each session. When the online banking server sees this final value, it can detect the code compromise.

```
01 var test = function(na, nb) {
02  var test2 = function(na, nb) {
03      var test3 = function(na, nb) {
04          var pre = na ^ crc32(arguments.callee.caller.toString());
05          var post = DomScan() ^ crc32(arguments.callee.caller.toString()) ^ nb;
06          return post;
07      };
08
09      var pre = na ^ crc32(arguments.callee.caller.toString());
10      var post = test3(pre) ^ crc32(arguments.callee.caller.toString()) ^ nb;
11      return post;
12  };
13
14  var pre = na ^ crc32(arguments.callee.caller.toString());
15  var post = test2(pre) ^ crc32(arguments.callee.caller.toString()) ^ nb;
16  return post;
17 };
18
19 (function(){
20  var na = 32053221, nb = 4321053;
21  result = test(na, nb);

22  console.log(result.toString(16));
23 })();
```

From the attacker's standpoint, dynamic inspection/replacement strategy could be quite tricky for this algorithm since extra handling for multi-phase inspection/replacement is difficult to correctly engineer. However, it's not impossible to crack this code. The robustness of this algorithm can be improved by chaining integrity check all along the pipeline. Moreover, traditional obfuscation techniques will add extra complexity in writing the bypass code.

Defense: Randomisation

But can't the attacker break it?

The answer to the question is a resounding, 'Yes'.

MIPS integrity check is a necessary step to ensure the operating environment is not compromised. However, this integrity check method can be defeated by a single smart algorithm once the attacker figures out how the algorithm works. What the attacker needs to do is to run regex, modify and reconstruct MIPS code to produce clean data after a

little bit of reverse engineering. Therefore, the demonstrated integrity check method itself is not sufficient to guarantee MIPS integrity.

One of the best strategies to tackle this problem is to randomise MIPS code. Defenders need to prepare a set of ***algorithmically heterogeneous*** MIPS code so that each online banking page request will return a different code that requires a dedicated attack algorithm. Another strategy is to split MIPS scripts into multiple files randomising the number of files, file name and splitting method. This makes it difficult for malware inject filter to identify, modify and reconstruct MIPS code.

Be reminded that the purpose of these randomisation strategies is to increase the reverse engineering cost in analysing different algorithms and developing the bypass code. It is best if randomisation algorithm can be designed and implemented with less cost than attackers need to invest to defeat each algorithm. Unfortunately it is likely that the attack algorithm development cost is linear with MIPS randomisation development cost. In other words, a randomisation algorithm that randomises many elements can be defeated by a single counter-algorithm instead of the total number of possible randomised elements. For example, a randomisation of nested closure based integrity check algorithm described above may be defeated by a single algorithm that performs regex, inspection code, replace injected code, and so on.

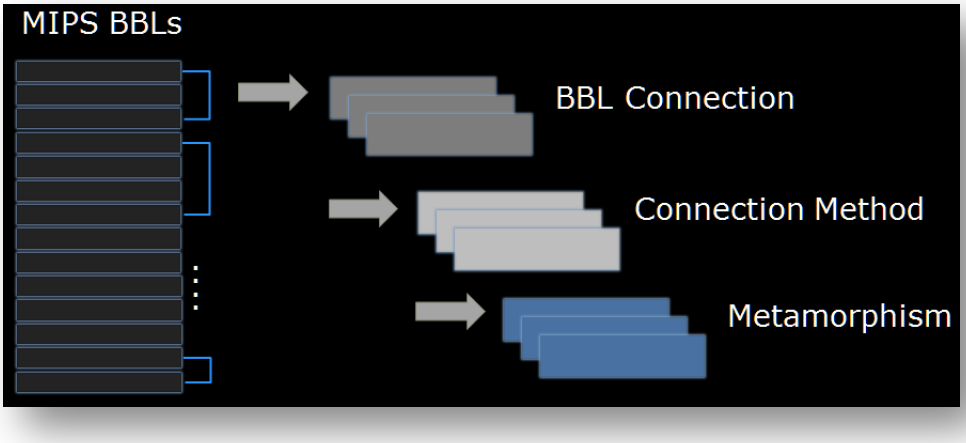
The following subsections introduce a couple of randomisation algorithms that attempt to push the attacker to create an algorithmically heterogeneous bypass code, which increases the attack cost.

Control Flow Randomisation

This method integrated MIPS integrity check algorithm into the control flow randomisation algorithm. The steps involved are as follows.

1. Create a single function that includes all MIPS code
2. Split it into arbitrary number and size of basic blocks in a way that it doesn't affect MIPS logic
3. Create a set of algorithms for the following techniques:
    a. Basic block connection mechanism (function, closure, exception, etc)
    b. Basic block connection mode (static or dynamic)
    c. Metamorphism (dead code, control flow arrangement, etc)
4. Create a set of randomised MIPS script instances by combination of step 2 and 3.

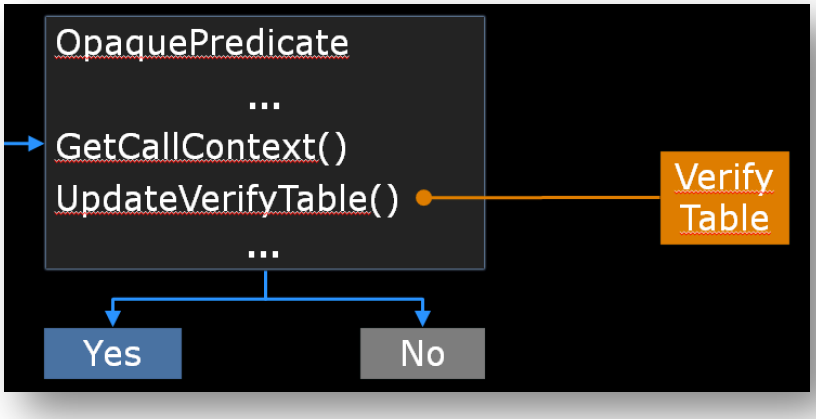
The above ranomisation process is depicted below.



Each individual randomisation approach is breakable in its own. Although the bypass algorithm can be quite complex, the attackers can simply chain individual method to defeat the randomisation. What poses a challenge to the attacker is the fact that the integrity check algorithm is spread within the randomised code.

Opaque Predicates

Opaque predicate refers to a complex branch code that always executes in one direction, which is known to the creator of the program, and which is unknown a priori to the analyser. If opaque predicate is used solely without supporting mechanism, the attacker does not need to spend the time dissecting it as long as attacker's bypass code works.



As with control flow randomisation, we set the challenge to the attacker by integrating MIPS integrity check deeply buried within the opaque predicate. We can put added complexity by merging integrity check data into part or all of MIPS intelligence data table. Any arbitrary algorithm can be used as an opaque predicate implementation as long as it produces predictable outcome (i.e. audio conversion, Fourier transform, genetic algorithm, etc)

In short, opaque predicate and code obfuscation is quite a challenging topic in academia. Check the pointers in the References section to explore this field.

Rootkit

Despite the efforts we made on MIPS integrity, MIPS is still vulnerable to MITM (Man-In-The-Middle) style attack. Malware can intercept user-defined JavaScript functions as well as JavaScript native code to tamper with the returned data. We introduce a term **DOM Rootkit** (or **JavaScript Rootkit**) due to the flow-wise similarity with the operating system world.

Attack: How It Works

One of the functions that malware may want to target is ***getElementsByTagName*** DOM function that MIPS may use to enumerate scripts running in the current browser session. By rootkitting this function, the attackers will give MIPS incorrect information. The following code snippet demonstrates a method that modifies the script enumeration result by hooking ***getElementsByTagName*** DOM function. Hooking in JavaScript is as simple as a single assignment operation with hook function body as shown below.

```
var original_func = document.getElementsByTagName;

document.getElementsByTagName = function () {
  var r = original_func.apply(document, arguments);
  for(var i=0; i<r.length; i++) {
    if(r[i].text.search(a_string_in_my_inject) != -1) {
      r[i].remove();
      console.log('Inject Rootkitted!');
      break;
    }
  }
  return r;
};
```

Focusing on script-hiding aspect, the following functions can be targeted, which need to be secured from defender's point of view.

Space	Function	Hiding Method
DOM	document.getElementsByTagName(selector)	Modify the returned HTMLCollection
jQuery	jQuery.find(selector, doc, ret)	Modify the returned array

Other functions critical to verify the DOM and MIPS integrity include the following:

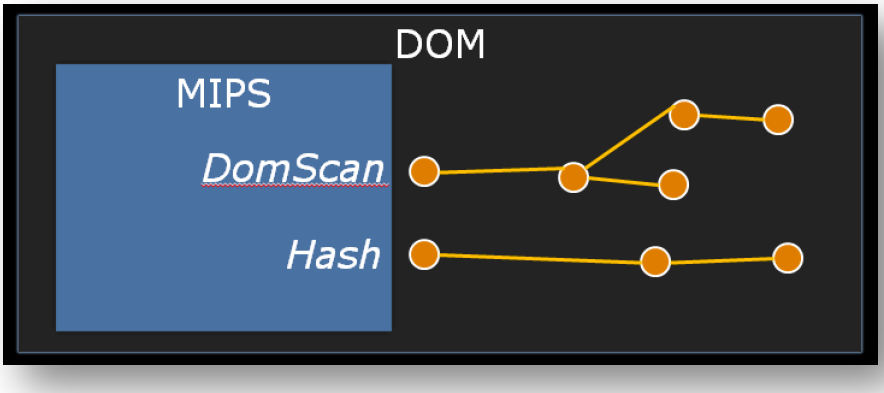
Function	Defender's Usage
Object.keys()	Retrieve all functions and variables in current DOM instance
Function.prototype.toString()	Retrieve the function source

Malware can hook critical MIPS functions such as ***DomScan(), Hash()*** and ***Salt()*** if they are implemented as a function.

Note that these lists are not comprehensive covering all attack vectors. All entry point and intermediate functions used by MIPS must be identified to check the presence of rootkit.

Defense: DOM Integrity Check

A straightforward defense for a DOM rootkit is a DOM integrity check. With this method, MIPS needs to collect signatures of all functions and their dependents that MIPS calls. Using the pre-calculated whitelist, MIPS system can find the DOM rootkit either on client side or on server side.



Defense: Detecting Rootkits

A DOM integrity check method seems to work. However, the result from a DOM integrity check can be forged by hooking DOM function. In JavaScript, not only user-defined functions but also native functions can be replaced by user functions. In addition, the traditional way of comparing before- and after-snapshots will not work because the browser environment is fully set by the malware inject before a single line of MIPS code gets executed.

The first line of defense is to cross-check the results returned from multiple execution paths. For instance, script nodes can be enumerated by walking through DOM elements from the root, using the selector with attributes, and so on.

However, this rootkit war is an arms race if you recall the rootkit war in the operating system world.

If you think out of the box, then there are always ways to detect rootkits. Exploiting exception handling mechanisms would be one of the promising methods to achieve this goal. For instance, ***Function.prototype.toString()*** is one of the critical functions that cannot afford to be compromised to perform our integrity check. Whether this function was hooked or not can be checked by deliberately injecting an exception to it. As the exception gets triggered within the JavaScript's native ***Function.prototype.toString()***, the stack trace on exception will give us all functions between our exception-triggering code and native code including the hooked function, if any.

The following code snippet injects ***TypeError*** exception.

```
01 var hooked = Function.prototype.toString;
02 Function.prototype.toString = function() {
03  console.log("YOU ARE ROOTKITTED!");
04  hooked.apply(this, arguments);
05 }
06
07 var TriggerException = function(){
08  try {
09      Function.prototype.toString.call('hooktest')
10  }
11  catch(err) {
12      console.log(err.stack);
13  }
14 }
15
16 TriggerException();
```

If a rootkit is present, the stack trace will contain the line highlighted in red as shown below.

```
TypeError: Function.prototype.toString is not generic
    at String.toString (native)
    at String.Function.toString (https://mybank.org/login?next=%2F:173:7)
    at TriggerException (https://mybank.org/login?next=%2F:177:29)
    at https://mybank.org/login?next=%2F:183:1
```

Again this is one of the techniques you can use combined with other rootkit detection methods. Creativity and diversity is the key for the survival in the rootkit war.

Fraud Analytics

MIPS fraud analytics system is a part of MIPS framework, which needs to be deployed on online banking server end. It collects MIPS intelligence, keeps the log in MIPS database, and performs detection. Apart from all analytics algorithms discussed in this white paper, there are several improvements that need to be made.

Attack: Blocking MIPS

MIPS is effective in detecting banking malware infected online banking sessions only if the bank gets MIPS intelligence. If the bank ignores missing MIPS intelligence, the attacker can evade MIPS by simply blocking MIPS related traffic.

Defense: MISSING_MIPS Event

It is ideal to implement MIPS as part of the core online banking application logic so that any fraudulent sign-in or transaction attempt can be processed by the banking business logic either in batch or in real time. If MIPS is not integrated into the online banking logic, the attackers could block MIPS message, which will leave MIPS fraud analytics system with no information to determine the presence of banking malware. Therefore, if it is not practical to integrate MIPS into the online banking application, MISSING-MIPS event should be implemented by MIPS fraud analytics system. First of all, a fraud analytics system needs to ensure MIPS intelligence is not cached by the proxy in transit. For instance you can put a random number in MIPS AJAX POST request to achieve this goal. And then, MISSING_MIPS event implementation is as straightforward as correlating online banking access log with MIPS log.

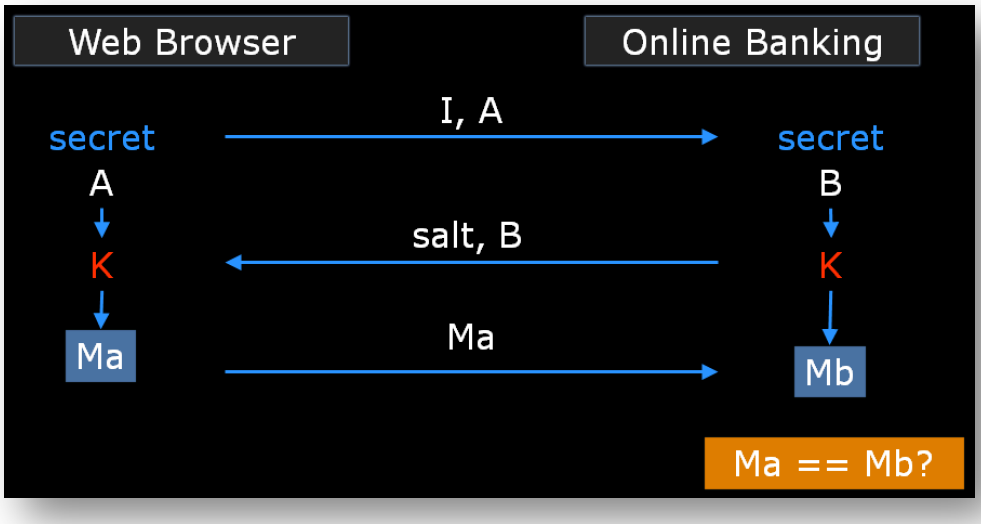
Detecting Moving Targets

Malware inject is just a piece of JavaScript software combined with HTML components. As with any software, the malware inject's feature won't change dramatically in each release. Once MIPS signature has been identified for the inject's source code, any minior change to it can be detected by locality sensitive hashing algorithms such as TLSH. Locality sensitive hashing is simply a hash for a string like standard one-way hash such as MD5 and SHA1. But its hash value changes slightly on the changed part of the string only, unlike standard one-way hash algorithms where a single byte of change produces a totally different hash value.

Using locality sensitive hashing in replacement of standard one-way hashing, MIPS can effectively detect evolving malware injects on minor upgrade.

Zero Knowledge Proof (ZKP)

Zero Knowledge Proof refers to a method that meets the property where two parties sharing the same secret can verify that they share a secret without transmitting the actual secret. Secure Remote Password (SRP) protocol is an implementation of ZKP. A simplified conceptual diagram of SRP protocol is shown below.



Note that this diagram depicts the concept only without showing the actual details. For a full detail, check the Wiki [7, 8, 9, 10].

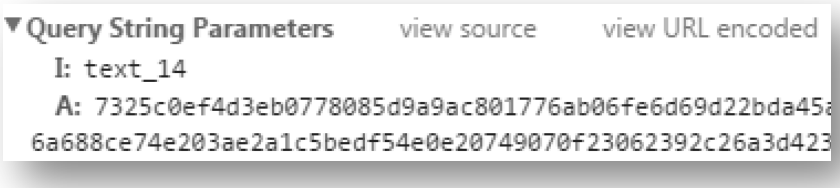
Assuming a shared secret has been already established, the client (web browser in the above diagram) initiates the protocol by sending ***I*** (identifier of the shared secret) and ***A*** (a number derived from a random number generated by the client). The server (online banking server in the above diagram) then replies back with ***B*** (a number derived from a random number generated by the server) and salt (per-session or per-secret random number). By this time both parties can derive a shared key, ***K***, from available information to each party. At this point, with a hash on all parameters available except the secret, both parties can discover whether or not they share the same secret with each other.

SRP on the wire

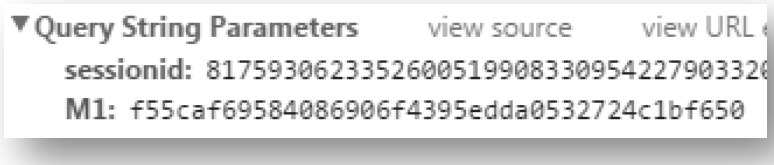
An example implementation of ZKP using SRP can be found in Appendix C. The following shows network packets that correspond to ZKP. ***/mips/zkp_start*** is an initiation of ZKP from the client and ***/mips/zkp_verify*** is the final stage of the protocol.

```
/mips/zkp_start?I=text_14&A=2ccaf4d78a5ad576907d7bbf17bba358f3...
/mips/zkp_verify?sessionid=134702719795903606669962005099901 62...
```

The following shows HTTPS GET request parameters for the first traffic. The client sends out ***I*** (identifier of the secret) and ***A*** (a random number generated by the client).



The following shows HTTPS GET request parameters for the second traffic, which contains ***sessionid*** received from the first traffic and ***M1***, which is a hash to prove the possession of the secret that will be compared on the server.



The key to this network analysis is that none of the transmitted data contains the shared secret while both the client and the server successfully verify the fact that they share the same secret.

Use Cases

In the simplest form, ZKP can be used to prevent any online banking secret from being transmitted over the wire This can include sign in password and OTP (One –Time Password) token for transaction authorisation.

Despite its cryptographic appeal, ZKP doesn't mean much within the banking malware threat landscape because the malware inject can simply grab the secret before ZKP protocol is initiated. However, ZKP deployment has the effect of forcing the attackers to engage with MIPS protocol since there are some attack groups who just passively sniff network traffic without injecting a malicious script on the page.

A more important usage of ZKP is MIPS hardening. Combined with all defense techniques presented in this white paper, ZKP can provide additional layer of MIPS protection. There are many pieces of DOM information collected by MIPS, most of which have static values that can be used as shared secrets. Examples of those ZKP-able candidates are as follows:

- DOM function integrity data
- MIPS integrity data
- MIPS rootkit detection data
- MIPS intelligence format

Especially a large chunk of collected DOM data can be effectively integrity-checked through ZKP. In addition, MIPS intelligence format can be disguised through ZKP. It not only hides format, but also removes the need to save MIPS log in the database, which reduces the database size, allowing the fraud analyst to focus on anomalies only.

Since ZKP is not the front-line security defense against malware, it doesn't compromise the online banking security entirely even if ZKP is not correctly implemented. However, cryptographic requirements must be carefully reviewed when embedding ZKP within MIPS to ensure security properties are met within MIPS context.

Mobile Banking

MIPS framework works as long as the online banking facility runs in the web browser environment. Therefore the customers using the web browsers in the mobile phone can be protected under MIPS framework. However, standalone mobile banking applications pose a different set of challenges since its operating environment is exposed to different style of attack vectors such as mobile banking app replacement and keyboard logging/swapping. Despite this incoherent threat environment, the core MIPS principles can be still applied.

How can MIPS framework be used?

First off, we need to bring the attackers in the same battle ground. This can be accomplished by designing the mobile banking applications in such a way that they follow MIPS protocol in order to be conceived as legitimate sign-in and transactions. This forces the mobile banking malware to engage with MIPS protocol. It allows the defenders to use the same techniques as browser-based online banking does such as code randomisation, MIPS integrity verification and rootkit detection.

Once the MIPS protocol becomes a part of mobile banking, the artefacts left by the mobile banking malware need to be collected. As the attack vectors are different in mobile banking environment, the defenders should collect artefacts associated with individual attack vectors instead of injected JavaScript.

Finally, MIPS integrity check can be applied to mobile app's Java code depending on how the app is designed.

Conclusion

Banks need to go through a rigid set of tests before any change is released, which can take up more time than malware's inject release cycle. In order to win the online banking war, it is crucial to prepare strategic defense rather than short term ad-hoc patching since overall update structure is not in favor of the defenders in the online banking war.

Diversity of implementation is the key for survival. Defenders need to be creative and out-smart the cybercriminals in the code war to win the battle. It is also very important to understand MIPS fraud analytics system is fully exposed to various attacks as MIPS intelligence is coming from the enemy's territory. Defenders must assume MIPS intelligence is potentially hostile and perform proper application security checks such as input validation and output encoding.

As a final note,

'Never explicitly block on the spot on detection! They will come back the next day!'

References

1. Ruxcon 2013: Inside Story Of Internet Banking: Reversing The Secrets Of Banking Malware
2. Dalla Preda, Mila. "Code obfuscation and malware detection by abstract interpretation." PhD diss.), http://profs. sci. univr. it/dallapre/MilaDallaPreda_PhD. pdf (2007).
3. Moser, Andreas, Christopher Kruegel, and Engin Kirda. "Limits of static analysis for malware detection." Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual. IEEE, 2007.
4. Oliver, J., Cheng, C., Chen, Y.: TLSH - A Locality Sensitive Hash. 4th Cybercrime and Trustworthy Computing Workshop, Sydney, November 2013, https://www.academia.edu/7833902/TLSH_-A_Locality_Sensitive_Hash
5. Oliver, J., Forman, S., and Cheng, C.: Using Randomization to Attack Similarity Digests. ATIS 2014, November, 2014, pages 199-210. https://www.academia.edu/9768744/On_Attacking_Locality_Sensitive_Hashes_and_Similarity_Digests
6. http://reverseengineering.stackexchange.com/questions/1669/what-is-an-opaque-predicate
7. https://en.wikipedia.org/wiki/Zero-knowledge_proof
8. https://en.wikipedia.org/wiki/Secure_Remote_Password_protocol
9. http://srp.stanford.edu/
10. http://code.google.com/p/srp-js/
11. http://reverseengineering.stackexchange.com/questions/1669/what-is-an-opaque-predicate
12. https://tampermonkey.net/

Appendix A

```
/*
 MIPS v1.

 v1 enumerates all script DOM elements and performs AJAX POST with SHA1 of script texts (after normalisation) or raw src string.
*/
function GetMipsData() {
 var mips = [];

 var i = 1;
    $("script").each(function(index){
     var type = '';
     var value = '';

     type = 'text';
        value = $(this).text();

     if(value != ''){
         console.log('['+i.toString()+'] ' + value);
         value = $.sha1($.normalise(value));
     }
     else{
         type = 'src';
            value = $(this).attr("src");
     }

     mips.push({
         "name": type + '_' + i.toString(),
         "value": value
     });

     i++;
    });

 var str =  JSON.stringify(mips);
 //console.log(str);

 $.ajax({
     url: "mips/v1?r=" + new Date().getTime() + "&_xsrf=" +  getCookie("_xsrf"),
     type: "POST",
     contentType: "application/json",
     async: false,
     data: Base64.encode(str)
 });
}

// Collect MIPS data on page unload
$(window).unload(function() {
 GetMipsData();
});
```

Appendix B

```
// Enumerate all user-defined functions/objects

function GetUserObjects(obj){
 return Object.keys(obj).filter(function(x){
     if (!(obj[x] instanceof Function))
         return false;

     return !/\[native code\]/.test(obj[x].toString()) ? true : false;
 });
}

function EnumUserObject(objname, obj){
 var names = GetUserObjects(obj);
 for (var i=0; i<names.length; i++){
     console.log("[" + objname + ":" + i.toString() + "] "+ names[i] + "\n>> " + obj[names[i]]);
 }
}

var names = GetUserObjects(window);
for (var i=0; i<names.length; i++){
 console.log("* " + names[i] + "\n>> " + window[names[i]]);
 EnumUserObject(names[i], window[names[i]]);
 console.log("---------------------------------------");
 EnumUserObject(names[i]+'.prototype', window[names[i]].prototype);
 console.log("=======================================");
}
```

Appendix C

The following code snippet is a demo SRP implementation. This is a client side JavaScript code. MIPS script can run SRP protocol by calling:

$.zkp(name, value, 'mips');

where name is an identifier for the secret (value) when there are multiple shared secrets.

```
function SRP(I, secret, url) {
  var srp = this;
  var I = I;
  var secret = secret;

  var N = new
BigInteger("c037c37588b4329887e61c2da3324b1ba4b81a63f9748fed2d8a410c2fc21b1232f0d3bfa024276cfd88448197a
ae486a63bfca7b8bf7754dfb327c7201f6fd17fd7fd74158bd31ce772c9f5f8ab584548a99a759b5a2c0532162b7b6218e8f142
bce2c30d7784689a483e095e701618437913a8c39c3dd0d4ca3c500b885fe3", 16);
  var g = new BigInteger("2");
  var k = new BigInteger("592b64cddab17ac6dc75a79c569637d05340f19d", 16);
  var r = new SecureRandom();

  var a = new BigInteger(32, r);
  var A = g.modPow(a, N);
  while(A.mod(N) == 0){
      a = new BigInteger(32, r);
      A = g.modPow(a, N);
  }

  this.H = function(arg) {
      return new BigInteger($.sha1(arg.join(':')), 16);
  }

  this.handshake1 = function() {
      return A.toString(16);
  };

  this.handshake2 = function(salt, Bstr) {
      var B = new BigInteger(Bstr, 16);
      var s = new BigInteger(salt, 16);

      // u = H(A,B)
         var u = srp.H([A, B]);

         // x = H(salt:I:secret))
         var x = srp.H([s, I, secret]);

      // S = (B - kg^x) ^ (a + ux)
         var kgx = k.multiply(g.modPow(x, N));
         var aux = a.add(u.multiply(x));
         var S = B.subtract(kgx).modPow(aux, N);
      //var Kstr = $.sha1(S.toString(16));
      var K = srp.H([S])

      // M1 = H(H(N) ^ H(g), H(I), s, A, B, K_c)
      var Ng = srp.H([N]).xor(srp.H([g]));
      M1 = srp.H([Ng, srp.H([I]), s, A, B, K]);
```

```
46
47      return  M1.toString(16);
48     };
49
50 }
51
52 (function($) {
53   $.extend({
54        zkp: function(I, secret, url){
55          var srp = new SRP(I, secret);
56
57          var Astr = srp.handshake1();
58          $.ajax({
59              url: url + "/zkp_start?" + "I=" + I + "&A=" + Astr + '&r=' + new Date().getTime(),
60              type: "GET",
61              async: false
62          }).done(function(body){
63
64              var M1 = srp.handshake2(body['s'], body['B']);
65              $.ajax({
66                  url: url + "/zkp_verify?" + 'sessionid=' + body['sessionid'] + '&M1=' + M1 + '&r='
+ new Date().getTime(),
67                  type: "GET",
68                  async: false
69              }).done(function(body){
70                  console.log('ZKP protocol complete');
71              });
72          });
73      }
74   });
75 })(jQuery);
```

The following is server-side code example implemented in Tornado Python. For demonstration purposes ZKP class in the code snippet has a hard-coded ***secrets*** dictionary that maps scanned JavaScript's ***type_index*** to its SHA1 hash value. Here two clean script hashes are stored in ***secrets*** dictionary for testing.

```
001 #!/usr/bin/python
002 #
003 # protocol.py
004
005 import random
006 from hashlib import sha1
007
008 """ I shamelessly borrowed some functions from the wiki:
009 https://en.wikipedia.org/wiki/Secure_Remote_Password_protocol
010 """
011
012 def H(*a):  # a one-way hash function
013     a = ':'.join([str(a) for a in a])
014     return int(sha1(a.encode('ascii')).hexdigest(), 16)
015
016 def cryptrand(n=1024):
017     return random.SystemRandom().getrandbits(n) % N
018
019
020 N = '''00:c0:37:c3:75:88:b4:32:98:87:e6:1c:2d:a3:32:
021         4b:1b:a4:b8:1a:63:f9:74:8f:ed:2d:8a:41:0c:2f:
022         c2:1b:12:32:f0:d3:bf:a0:24:27:6c:fd:88:44:81:
023         97:aa:e4:86:a6:3b:fc:a7:b8:bf:77:54:df:b3:27:
024         c7:20:1f:6f:d1:7f:d7:fd:74:15:8b:d3:1c:e7:72:
025         c9:f5:f8:ab:58:45:48:a9:9a:75:9b:5a:2c:05:32:
026         16:2b:7b:62:18:e8:f1:42:bc:e2:c3:0d:77:84:68:
027         9a:48:3e:09:5e:70:16:18:43:79:13:a8:c3:9c:3d:
028         d0:d4:ca:3c:50:0b:88:5f:e3'''
```

```
029 N = int(''.join(N.split()).replace(':', ''), 16)
030 g = 2
031 k = H(N, g)
032
033
034 class ZKP(object):
035     def __init__(self):
036         self.sessions = {}
037         self.secrets = {
038             'text_14': '4fd62c9150baf6e12bef17efe03361a7995a5b01',
039             'text_15': 'dbe26126851643a6867d6a03da80b6a428f38c85'
040         }
041
042     def GetSessionId(self, n=256):
043         return str(random.SystemRandom().getrandbits(n))
044
045     def GetSalt(self):
046         return cryptrand(64)
047
048     def GetSecret(self, I):
049         if I in self.secrets:
050             return self.secrets[I]
051         else:
052             return None
053
054     def Start(self, I, A):
055         sessionid = self.GetSessionId()
056         salt = self.GetSalt() # per-session salt
057
058         A = int(A, 16)
059         if A % N == 0:
060             print 'ZKP: invalid A'
061             return { 'sessionid': '', 's': '', 'B': '' }
062
063         # Get secret
064         secret = self.GetSecret(I)
065         if secret == None:
066             print 'ZKP: invalid secret -> ' + I
067             return { 'sessionid': '', 's': '', 'B': '' }
068
069         # Calculate verifyer
070         # Note: Use saved verifier v and salt for speed improvement
071         x = H(salt, I, secret) # H(int, str, str)
072         v = pow(g, x, N)
073
074         # Calculate B and u till modulo requirement is satisfied.
075         while True:
076             b = cryptrand(32)
077             B = (k * v + pow(g, b, N)) % N
078             u = H(A, B)     # H(int, int)
079             if B % N != 0 and u % N != 0:
080                 break
081
082         # Calculate session key
083         S = pow(A*pow(v,u,N), b, N)
084         K = H(S)    # H(int)
085
086         M2 = H(H(N)^H(g), H(I), salt, A, B, K)
087
088         self.sessions[sessionid] = {
089             'M2': hex(M2)[2:-1]
090             }
091
092         # Javascript is unable to handle raw big integer.
093         return { 'sessionid': sessionid, 's': hex(salt)[2:-1], 'B': hex(B)[2:-1] }
```

```
094
095     def Verify(self, sessionid, M1):
096         if sessionid in self.sessions:
097             session = self.sessions[sessionid]
098 
099             M2 = self.sessions[sessionid]['M2']
100             print 'M1 = ' + M1
101             print 'M2 = ' + M2
102 
103             if M1 == M2:
104                 print 'Success'
105             else:
106                 print 'Fail'
```