SANDBOXING

NATIVE CLIENT

PRESENTED BY : HAMZA OMAR

OUTLINE

1. What is a Sandbox ?
2. Google Native Client
3. Demo for Native Client
4. Symbolic Execution
5. Demo for Symbolic Execution

WHAT IS A SANDBOX?

- In computer security, a **sandbox** is a security mechanism for separating running programs. It is often used to execute untested code, or untrusted programs from unverified third parties, suppliers, untrusted users and untrusted websites.
- A sandbox typically provides a tightly controlled set of resources for guest programs to run in, such as scratch space on disk and memory
- Sandboxing is frequently used to test unverified programs that may contain a virus or other malicious code, without allowing the software to harm the host device.

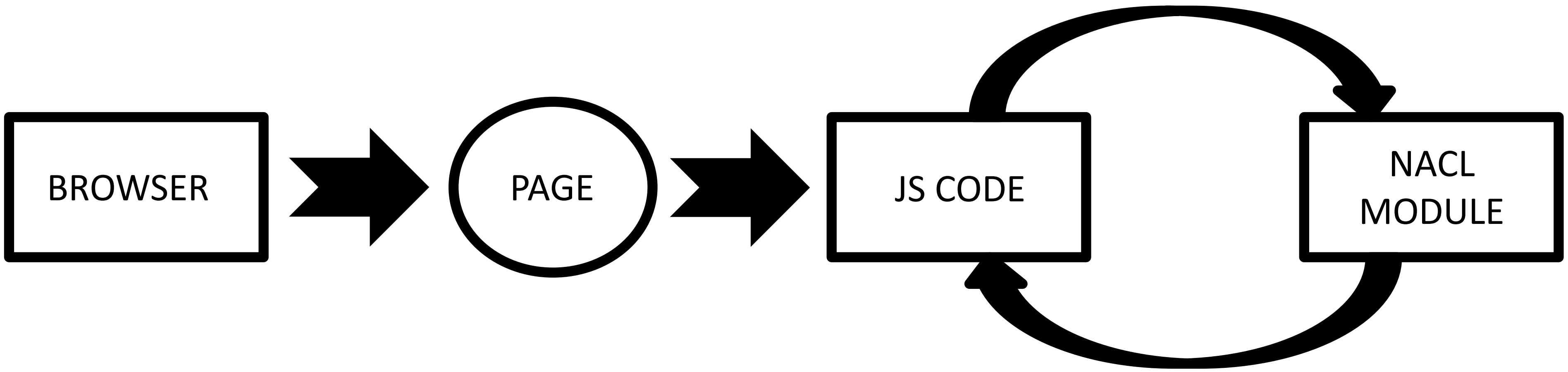
NATIVE CLIENT

- **Google Native Client** (**NaCl**) is a sandboxing technology for running a subset of Intel x86/x86-64, ARM or MIPS native code in a sandbox. It allows safely running native code from a web browser, independent of the user operating system, allowing web-based applications to run at near-native speeds.
- Real world system deployed by **GOOGLE** in their **CHROME** browser.
- It uses arbitrary native code with the help of isolation, sandboxing technique also called software fault isolation.
- **Software Fault Isolation** does not rely on the operating system to sandbox instead it looks at the binaries through a different approach to check whether it is safe to use the code or not.

WHY NATIVE CODE ?

- Web browsers already support JavaScript, flash or many others of kind. Why do we need native code then ?
- **PERFORMANCE -** The native code is unsafe from some perspectives but is really fast.
- **LEGACY CODES –** Not everything is written in JavaScript, so if we have an existing code that we want to run on a web application. No need for re-implementing.
- **SUPPORT FOR OTHER LANGUAGES –** If we don’t want to use JavaScript, we can use some other languages like C, C++, Python etc.

WHAT IS GOING ON !!



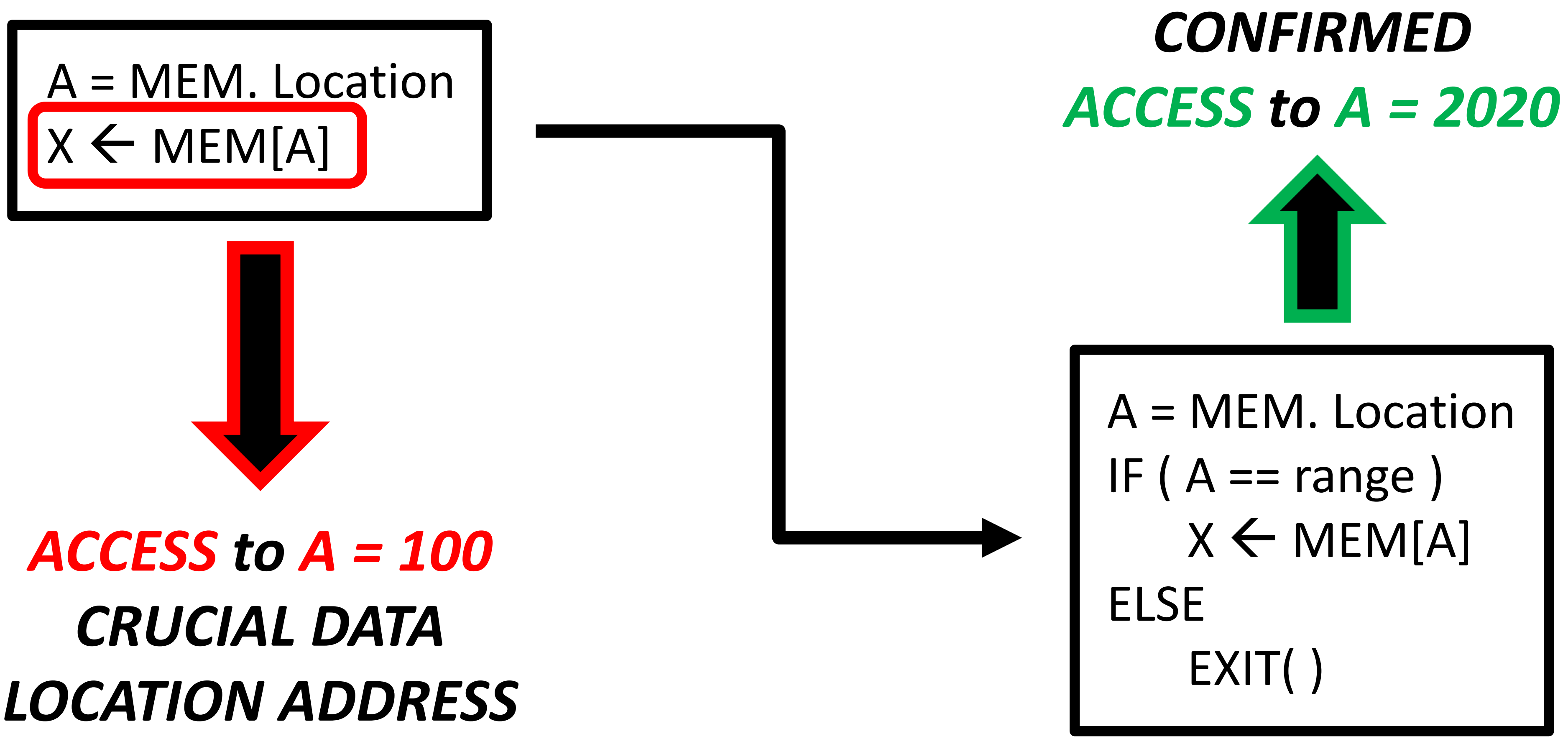
SECURING NATIVE CODE

- Trust the developer or ask the user whether they want to run a piece of code in their browser or not.
  - There might be a case, where the website asks the user to open up a page, and if user says yes, the page gets re-directed and crashes the browser.
  - Active X by Microsoft etc.
  - **Native Client gives the guarantee that if you run this program, no harm will come to you.**
  - This gives users the confidence to trust.
- OS/HW isolation.
  - Write a code in a OS that ensures isolation and sandboxing, like UNIX, capsicum etc.
  - There can be OS bugs and some OS are not compatible with each other.
  - We have to worry about what code we actually write inside a sandbox and it’s compatibility with the OS.
  - **Native Client does not worry about these problems because it runs the same code which runs on any OS.**

SOFTWARE FAULT ISOLATION

- The plan is actually **not to rely on the OS to check the code** at the run time rather look ahead of time that this code is safe or not for the system.
- Check ahead of time the binaries of the instructions whether these are safe instructions or un-safe.
- If instructions are **SAFE**, just allow them to pass.
  - What are actually safe instructions ? ALU, some mathematic instructions, move etc.
  - Do computations on its own little memory, meaning it can not access the disk, can not access the network etc.
- If instructions are **UNSAFE**, either **instrument the instruction** or prohibit it.
  - Instrumenting instruction means to introduce for example some checks before an access. By this we do not rely on the OS.
  - Unsafe instructions are actually memory accesses, instructions which could invoke a system call to switch privilege levels etc.
- When we are done with checking, we can run the program and by definition it will not do bad things to our system.

INSTRUMENTING INSTRUCTIONS



TRUSTED SERVICE RUNTIME

- Now the code within the sandbox is safe, it will run absolutely fine. It would not access the disk, will not access the browser, the display, will not access the network etc. instead it would just work on its own little chunk of resources allocated.
- Even if the code somehow, does not function in a safe way, a special service code has been provided by **GOOGLE** which is called Trusted Service Runtime.
- TSR actually gives the final assurance that the code is now safe.
- When the code inside the sandbox wants to allocate memory, spawn threads, or communicate to the browser, etc. It actually sends a call to the TSR and TSR does all things for that code.

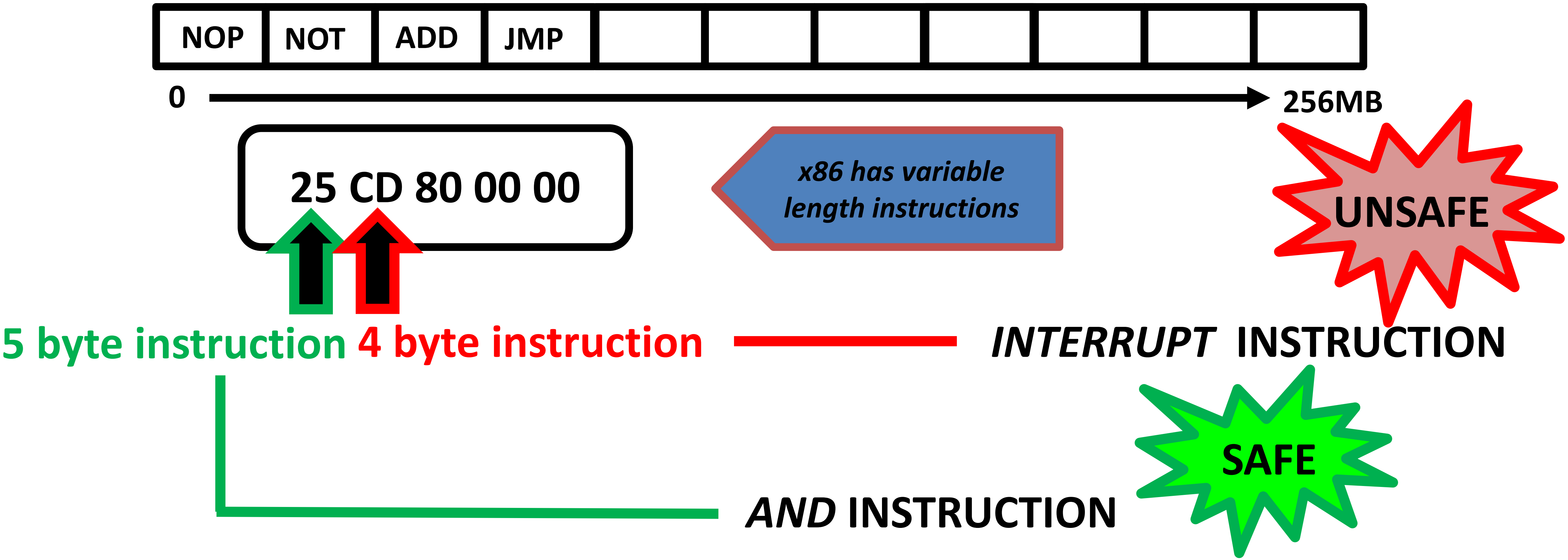
SAFETY

- For a native client safety means;

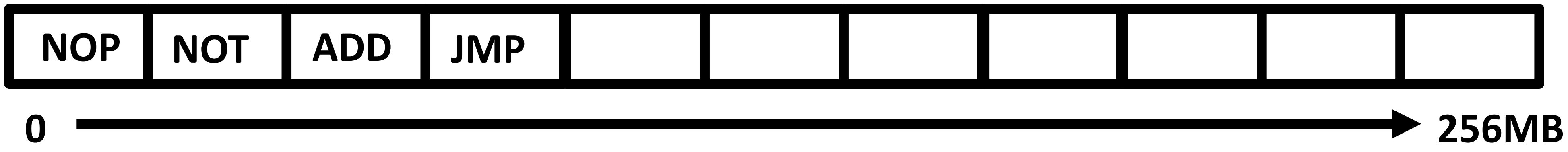
  1. No disallowed instructions are going to execute like system calls etc.
  2. No disallowed calls to jump out of the sandbox.
  3. All code + data accesses are in bounds for the module.

Usually a space of **0** to **256 MB** is provided to the module. So all the work must be with in this address space.

NAIVE APPROACH FOR SAFETY



RELIABLE DISASSEMBLY – NATIVE CLIENT



If we have a JUMP instruction in the application, and it jumps to a point which we did not see during our left to right scan by the core. **NOW WHAT ?**

If we have a JUMP instruction in the application, we will check whether the destination has been seen before if YES then OK otherwise the jump is INVALID.

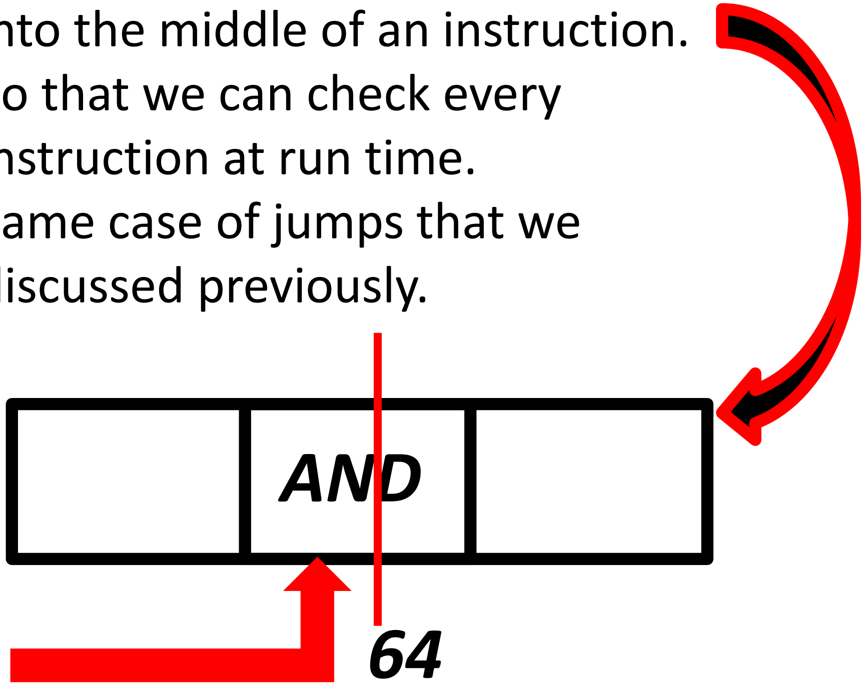
RULES

1. NACL's plan relies on binaries, if perturbed their system is broken thus their prevent writing to the binary once loaded.
2. They just have set a standard for simplicity.
3. These indirect jumps must be instrumented and checked for the target. Also, they should jump in multiples of 32.
4. In order to terminate the module as soon as the instructions are completed. And no jump instruction causes the core to jump out of the boundary of the module.

C1	Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
C2	The binary is statically linked at a start address of zero, with the first byte of text at 64K.
C3	All indirect control transfers use a `nacljmp` pseudo-instruction (defined below).
C4	The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4).
C5	The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
C6	All *valid* instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
C7	All direct control transfers target valid instructions.

RULES

5. Every multiple of 32 must be a valid instruction otherwise we will jump into the middle of an instruction.
6. So that we can check every instruction at run time.
7. Same case of jumps that we discussed previously.



C1	Once loaded into the memory, the binary is not writable, enforced by OS-level protection mechanisms during execution.
C2	The binary is statically linked at a start address of zero, with the first byte of text at 64K.
C3	All indirect control transfers use a `nacljmp` pseudo-instruction (defined below).
C4	The binary is padded up to the nearest page with at least one `hlt` instruction (0xf4).
C5	The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary.
C6	All *valid* instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
C7	All direct control transfers target valid instructions.

SEGMENTATION

- Whenever a processor is running there is a table maintained by the hardware called segment descriptor table. It has a bunch of segments having two values **BASE** and **LENGTH.**
- It shows that for a segment we have a chunk of memory which starts from the base address and ends at base + length.
- Instructions when accessing memory always point to a specific segment depending on the usage.
- Code segment, data segment, stack segment etc. discussed in the SGX slides.
- For example, MOV [A] [B]. A and B will used by the data segment which will take it's respective base address from the SGT and move the addresses.

So we need to prohibit MOV instructions in such a way that once a segment register gets a value it cannot be changed.

TRAMPOLINE - SPRINGBOARDS

Trampoline:

- For instructions requiring use of segment registers, native client module uses jumps into a special address space called a trampoline.
- In the trampoline, it performs all the operations using segment registers and segment descriptor table and jumps back to its module.

Springboards:

- When the segments registers are set by the instruction from the untrusted module, the trusted code first HALTS until the values are consumed.
- When done, it resets the values of segment registers back to their original states.

NaCL DEMO

Fri, 22 Apr 17:08 hao15101

Terminal

File Edit View Terminal Go Help

```
      } else {
        // It's possible that the Native Client module onload event fired
        // before the page's onload event.  In this case, the status message
        // will reflect 'SUCCESS', but won't be displayed.  This call will
        // display the current message.
        updateStatus();
      }
    }

    // Set the global status message.  If the element with id 'statusField'
    // exists, then set its HTML to the status message as well.
    // opt_message The message test.  If this is null or undefined, then
    // attempt to set the element with id 'statusField' to the value of
    // |statusText|.
    function updateStatus(opt_message) {
      if (opt_message)
        statusText = opt_message;
      var statusField = document.getElementById('statusField');
      if (statusField) {
        statusField.innerHTML = statusText;
      }
    }
  </script>
</head>
<body onload="pageDidLoad()">

  <h1>NaCl C++ Tutorial: Getting Started</h1>
  <p>
    <!--
    Load the published pexe.
    Note: Since this module does not use any real-estate in the browser, its
    width and height are set to 0.

    Note: The <embed> element is wrapped inside a <div>, which has both a 'load'
    and a 'message' event listener attached.  This wrapping method is used
    instead of attaching the event listeners directly to the <embed> element to
    ensure that the listeners are active before the NaCl module 'load' event
    fires.  This also allows you to use PPB_Messaging.PostMessage() (in C) or
    pp::Instance.PostMessage() (in C++) from within the initialization code in
    your module.
    -->
    <div id="listener">
      <script type="text/javascript">
        var listener = document.getElementById('listener');
        listener.addEventListener('load', moduleDidLoad, true);
        listener.addEventListener('message', handleMessage, true);
      </script>

      <embed id="hello_tutorial"
             width=0 height=0
             src="hello_tutorial.nmf"
             type="application/x-pnacl" />
    </div>
  </p>

  <h2>Status <code id="statusField">NO-STATUS</code></h2>
</body>
</html>
                                                            37,7          Bot
```

Fri, 22 Apr 17:09 hao15101

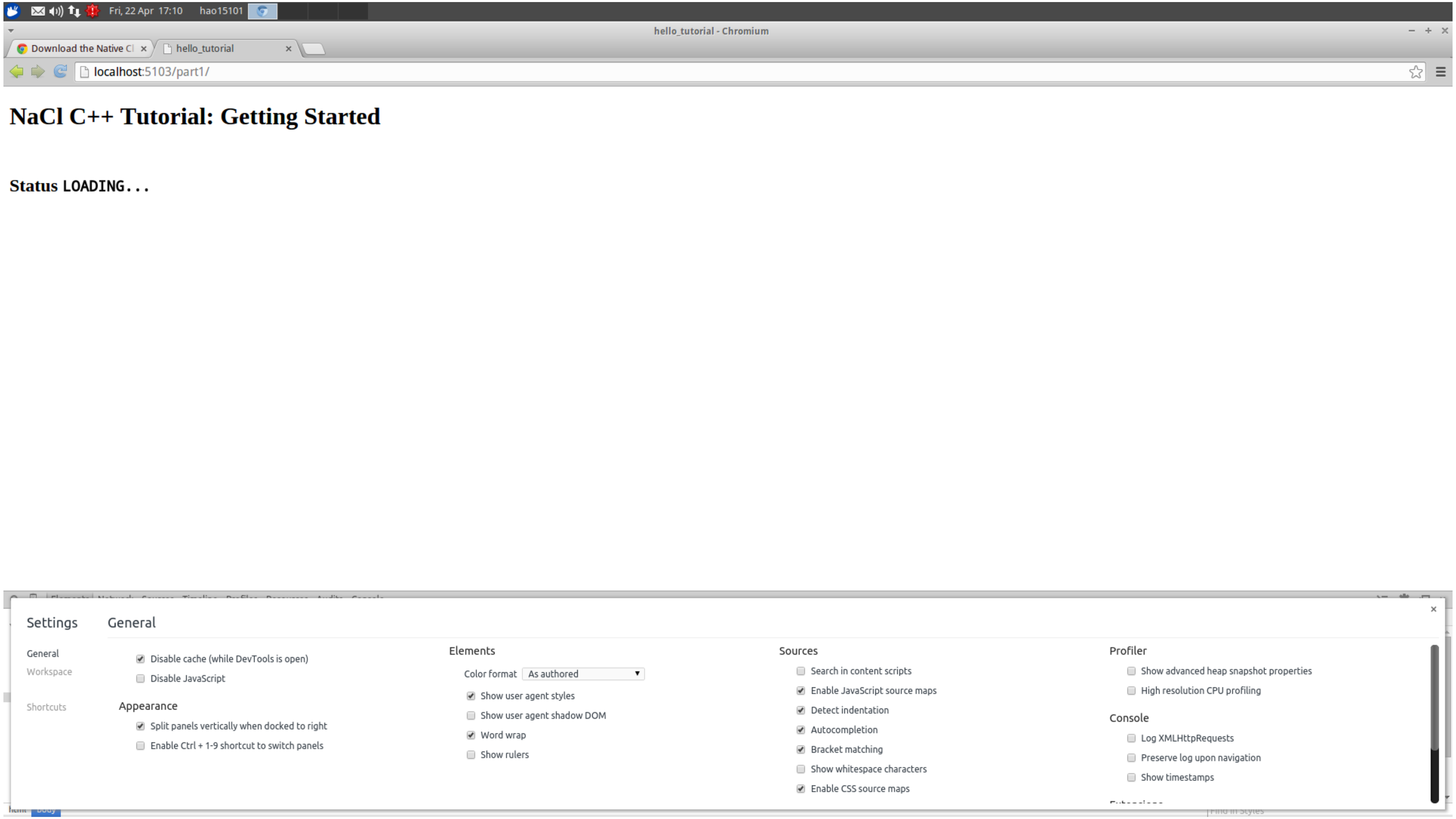
Terminal

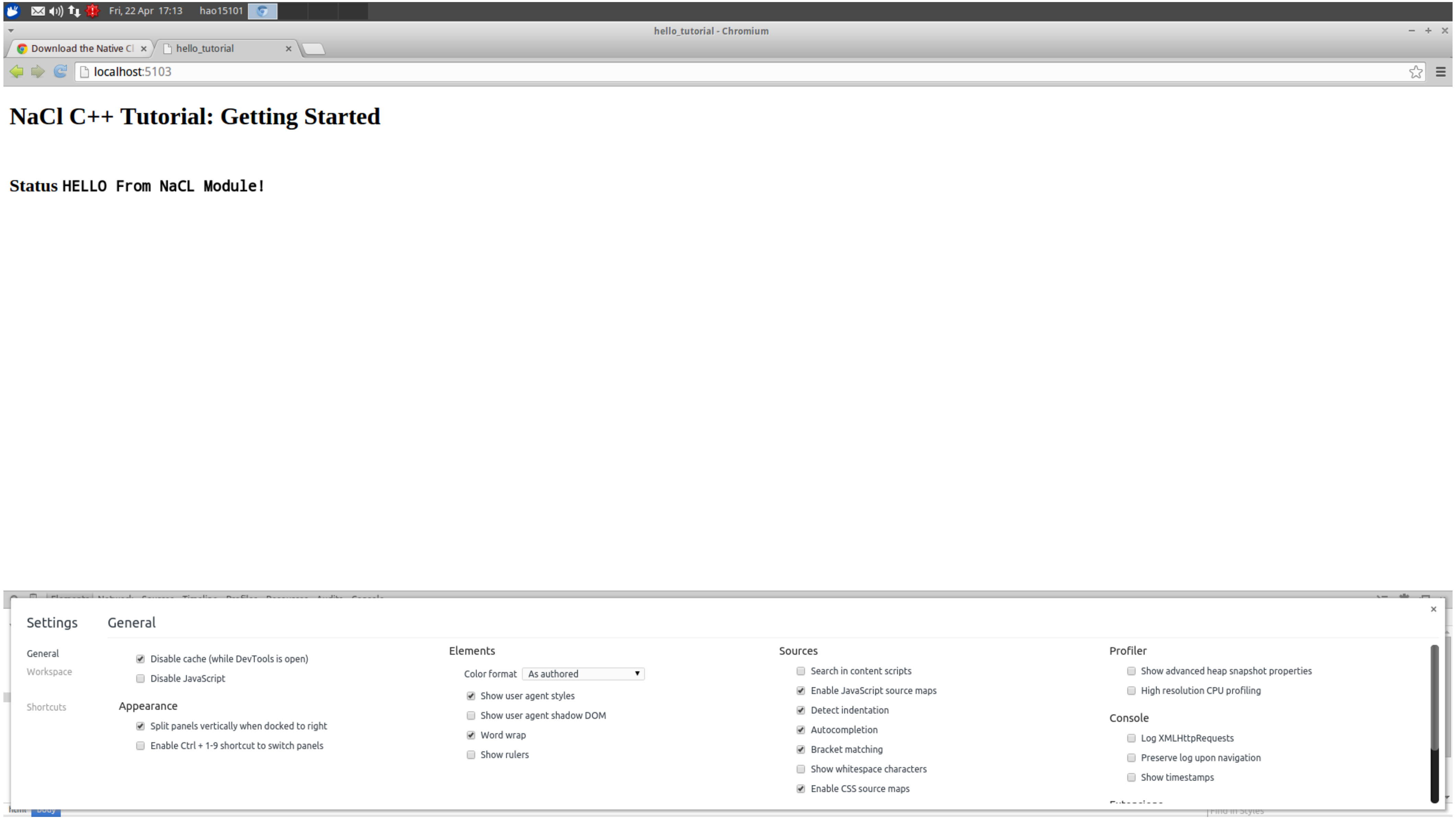
File Edit View Terminal Go Help

```
<!DOCTYPE html>
<html>
  <!--
  Copyright (c) 2013 The Chromium Authors. All rights reserved.
  Use of this source code is governed by a BSD-style license that can be
  found in the LICENSE file.
  -->
<head>

  <title>hello_tutorial</title>

  <script type="text/javascript">
    HelloTutorialModule = null;  // Global application object.
    statusText = 'NO-STATUS';

    // Indicate load success.
    function moduleDidLoad() {
      HelloTutorialModule = document.getElementById('hello_tutorial');
      updateStatus('SUCCESS');
      HelloTutorialModule.postMessage('hello');
    }

    // The 'message' event handler.  This handler is fired when the NaCl module
    // posts a message to the browser by calling PPB_Messaging.PostMessage()
    // (in C) or pp::Instance.PostMessage() (in C++).  This implementation
    // simply displays the content of the message in an alert panel.
    function handleMessage(message_event) {
      alert(message_event.data);
    }

    // If the page loads before the Native Client module loads, then set the
    // status message indicating that the module is still loading.  Otherwise,
    // do not change the status message.
    function pageDidLoad() {
      if (HelloTutorialModule == null) {
        updateStatus('LOADING...');
      } else {
        // It's possible that the Native Client module onload event fired
        // before the page's onload event.  In this case, the status message
        // will reflect 'SUCCESS', but won't be displayed.  This call will
        // display the current message.
        updateStatus();
      }
    }

    // Set the global status message.  If the element with id 'statusField'
    // exists, then set its HTML to the status message as well.
    // opt_message The message test.  If this is null or undefined, then
    // attempt to set the element with id 'statusField' to the value of
    // |statusText|.
    function updateStatus(opt_message) {
      if (opt_message)
        statusText = opt_message;
      var statusField = document.getElementById('statusField');
      if (statusField) {
        statusField.innerHTML = statusText;
      }
    }
```

38,9 Top

```cpp
/// method on the object returned by CreateModule().  It calls CreateInstance()
/// each time it encounters an <embed> tag that references your NaCl module.
///
/// The browser can talk to your NaCl module via the postMessage() Javascript
/// function.  When you call postMessage() on your NaCl module from the browser,
/// this becomes a call to the HandleMessage() method of your pp::Instance
/// subclass.  You can send messages back to the browser by calling the
/// PostMessage() method on your pp::Instance.  Note that these two methods
/// (postMessage() in Javascript and PostMessage() in C++) are asynchronous.
/// This means they return immediately - there is no waiting for the message
/// to be handled.  This has implications in your program design, particularly
/// when mutating property values that are exposed to both the browser and the
/// NaCl module.

#include "ppapi/cpp/instance.h"
#include "ppapi/cpp/module.h"
#include "ppapi/cpp/var.h"

namespace{
   const char* const kHelloString = "hello";
   const char* const kReplyString = "hello from NaCl";
}

/// The Instance class.  One of these exists for each instance of your NaCl
/// module on the web page.  The browser will ask the Module object to create
/// a new Instance for each occurrence of the <embed> tag that has these
/// attributes:
///     src="hello_tutorial.nmf"
///     type="application/x-pnacl"
/// To communicate with the browser, you must override HandleMessage() to
/// receive messages from the browser, and use PostMessage() to send messages
/// back to the browser.  Note that this interface is asynchronous.
class HelloTutorialInstance : public pp::Instance {
 public:
  /// The constructor creates the plugin-side instance.
  /// @param[in] instance the handle to the browser-side plugin instance.
  explicit HelloTutorialInstance(PP_Instance instance) : pp::Instance(instance)
  {}
  virtual ~HelloTutorialInstance() {}

  /// Handler for messages coming in from the browser via postMessage().  The
  /// @a var_message can contain be any pp:Var type; for example int, string
  /// Array or Dictinary. Please see the pp:Var documentation for more details.
  /// @param[in] var_message The message posted by the browser.
  virtual void HandleMessage(const pp::Var& var_message) {
    // TODO(sdk_user): 1. Make this function handle the incoming message.
    if(!var_message.is_string())
       return;
    std::string message = var_message.AsString();
    pp::Var var_reply;
    if(message == kHelloString)
    {
       var_reply = pp::Var(kReplyString);
       PostMessage(var_reply);
    }

  }
};
"hello_tutorial.cc" 97L, 4279C                                    69,1          30%
```


Fri, 22 Apr 17:10
hao15101
hello_tutorial - Chromium
Download the Native Cl
hello_tutorial
localhost:5103/part1/
NaCl C++ Tutorial: Getting Started
Status LOADING...
Settings
General
General
Workspace
Shortcuts
Disable cache (while DevTools is open)
Disable JavaScript
Appearance
Split panels vertically when docked to right
Enable Ctrl + 1-9 shortcut to switch panels
Elements
Color format
As authored
Show user agent styles
Show user agent shadow DOM
Word wrap
Show rulers
Sources
Search in content scripts
Enable JavaScript source maps
Detect indentation
Autocompletion
Bracket matching
Show whitespace characters
Enable CSS source maps
Profiler
Show advanced heap snapshot properties
High resolution CPU profiling
Console
Log XMLHttpRequests
Preserve log upon navigation
Show timestamps


Fri, 22 Apr 17:13 hao15101
hello_tutorial - Chromium
Download the Native Cl
hello_tutorial
localhost:5103
NaCl C++ Tutorial: Getting Started
Status HELLO From NaCL Module!
Settings
General
General
Workspace
Shortcuts
Disable cache (while DevTools is open)
Disable JavaScript
Appearance
Split panels vertically when docked to right
Enable Ctrl + 1-9 shortcut to switch panels
Elements
Color format As authored
Show user agent styles
Show user agent shadow DOM
Word wrap
Show rulers
Sources
Search in content scripts
Enable JavaScript source maps
Detect indentation
Autocompletion
Bracket matching
Show whitespace characters
Enable CSS source maps
Profiler
Show advanced heap snapshot properties
High resolution CPU profiling
Console
Log XMLHttpRequests
Preserve log upon navigation
Show timestamps

SYMBOLIC EXECUTION

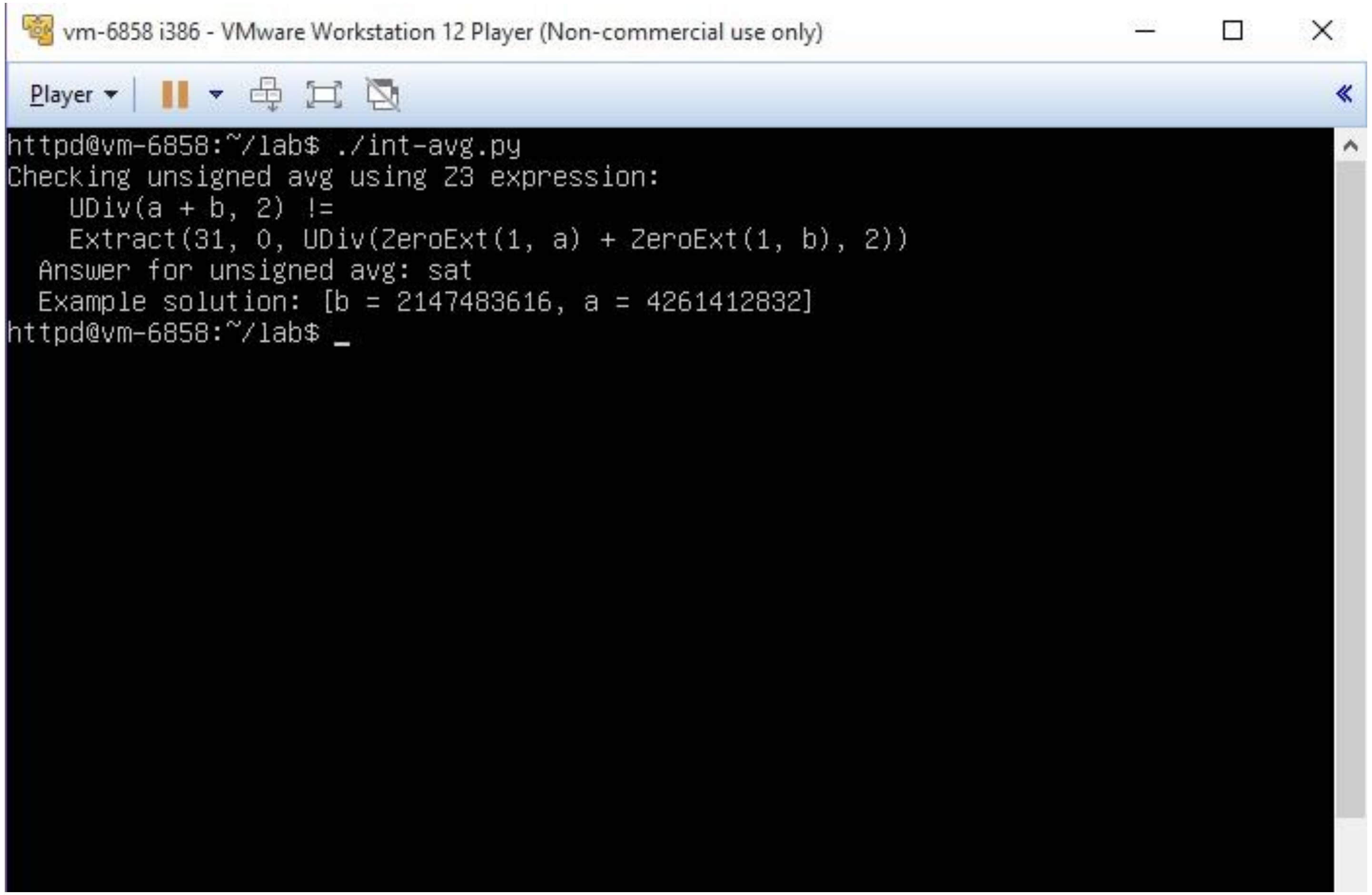
- Symbolic Execution is a mean of analyzing a program to determine what inputs cause each part of a program to execute.

IF will execute for y == 6

ELSE will execute for y != 6

```
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    fail();
  } else {
    printf("OK");
  }
}
```

SYMBOLIC EXECUTION



SYMBOLIC EXECUTION