Taint-enabled Reverse Engineering Environment (TREE)

Design Document

Revision History:

|  |  |  |  |
| --- | --- | --- | --- |
| Revision Number | Description | Authors | Timestamp |
| V0.1 | Initial Draft | Nathan Li,  Xing Li,  Loc Nguyen | 06/27/2013 |
| V0.2 | Update | Xing Li | 07/09/2013 |

Abstract:

This document describes the overall idea and architecture of Taint-enabled Reverse Engineering Environment (TREE) and detailed designs of its main components: TREE Tracer, TREE Analyzer and TREE Visualizer. The development of TREE is changing rapidly, and this document expects to be updated frequently.

This document is for users who intend to learn the designs and internals behind the tool. For installation, functionality and usage of the tool, please check for “TREE User Manual Document”.

Table of Contents

[2 TREE System Overview 3](#_Toc361153493)

[3 TREE Tracer 3](#_Toc361153494)

[3.1 Overview of Tracing 3](#_Toc361153495)

[3.2 Tracing with debuggers 3](#_Toc361153496)

[3.3 Tracing with system tracers 4](#_Toc361153497)

[3.4 Tracing with instrumentation tools 4](#_Toc361153498)

[3.5 Tracing with TREE 4](#_Toc361153499)

[3.5.1 TREE Tracer Plugin 4](#_Toc361153500)

[3.5.2 Saving configuration settings 5](#_Toc361153501)

[3.5.3 Run 5](#_Toc361153502)

[3.5.4 Attach 6](#_Toc361153503)

[3.6 Trace Generation with the TREE Tracer 6](#_Toc361153504)

[3.6.1 Using the PinAgent 6](#_Toc361153505)

[3.6.2 Using the IDA Debugger 6](#_Toc361153506)

[3.6.3 Interactive Mode 6](#_Toc361153507)

[3.7 IDA Pro’s Debugging framework 9](#_Toc361153508)

[4 TREE Analyzer 10](#_Toc361153511)

[4.1 Overview of Taint Analysis 10](#_Toc361153512)

[4.2 Main steps of Taint Analysis at Binary Level 10](#_Toc361153513)

[4.3 Taint Analysis in TREE 10](#_Toc361153514)

[4.4 Trace Parsing and Initial Taint Source Marking 10](#_Toc361153515)

[4.5 Taint Tracking 11](#_Toc361153516)

[4.5.1 X86 instruction static taint template 11](#_Toc361153517)

[4.5.2 Taint Precision: Taint by byte 12](#_Toc361153518)

[4.5.3 Taint Correctness: Over-Taint and Under-Taint 13](#_Toc361153519)

[4.6 Taint Checking and Security Analysis 13](#_Toc361153520)

[5 TREE Taint Analysis Testing Framework 13](#_Toc361153521)

[5.1 Overview of the Testing Framework: 13](#_Toc361153522)

[A working template: BasicOV 14](#_Toc361153523)

[5.2 Transformation function and the Oracle: 15](#_Toc361153524)

[5.3 X86 Instruction Coverage and the Priority List 15](#_Toc361153525)

[6 Test Suite: 16](#_Toc361153526)

[6.1 Code snippet (transformation-function like) unit level test suites for above instruction categories: 16](#_Toc361153527)

[6.2 Real world programs 16](#_Toc361153528)

[7 TREE Visualizer 17](#_Toc361153529)

[7.1 Overview of TREE Visualizer 17](#_Toc361153530)

[7.2 Graph Drawing 17](#_Toc361153531)

[7.3 Data Design 17](#_Toc361153532)

[7.4 Architecture 17](#_Toc361153533)

[7.5 TREE Visualizer Widget 17](#_Toc361153534)

[7.6 Interface 18](#_Toc361153535)

[7.6.1 IDA Pro Grapher 18](#_Toc361153536)

[8 References 18](#_Toc361153537)

# TREE System Overview

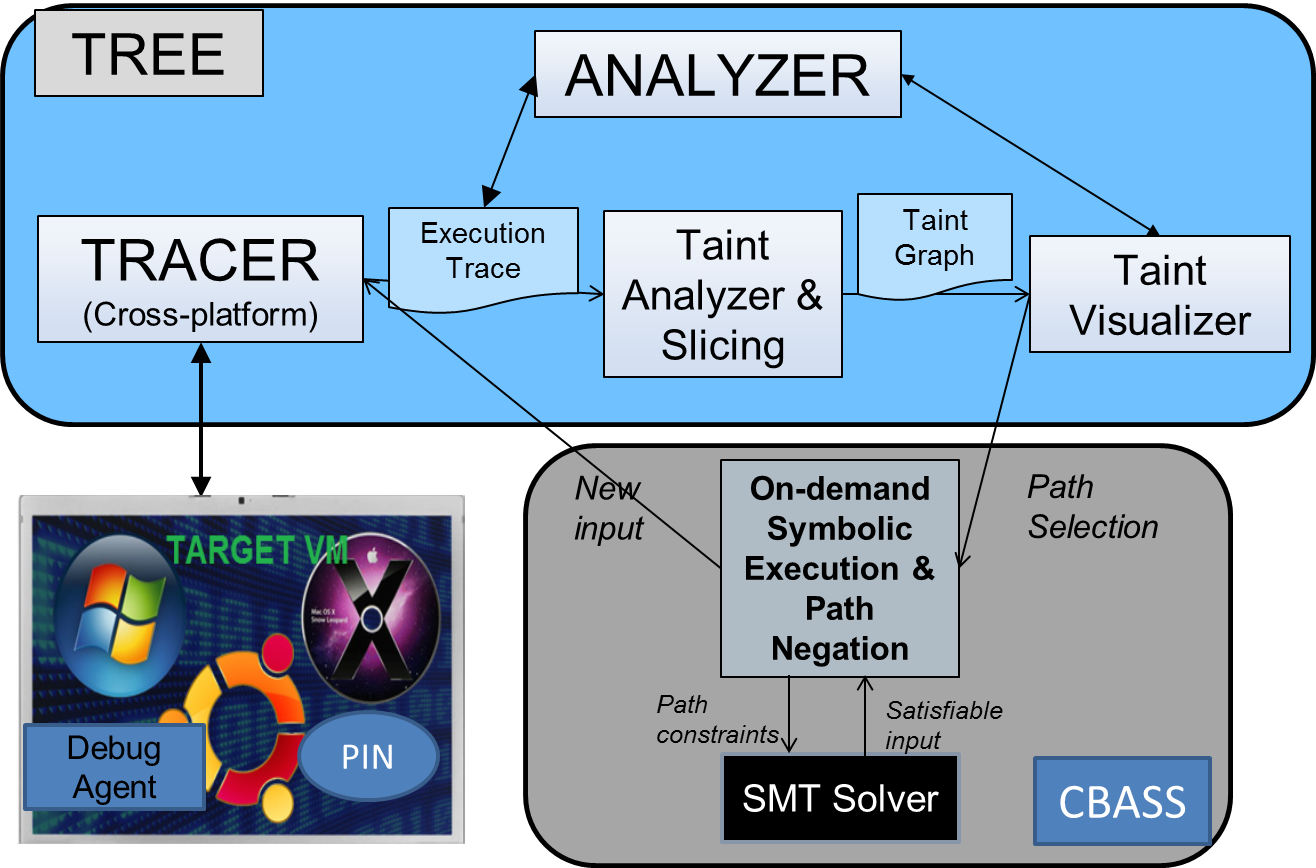


Figure 1 - TREE System Architecture

# TREE Tracer

## Overview of Tracing

Program tracing is very useful for programmers to debug and log a program’s execution. Detail instruction tracing can help programmers locate the exact location of a crash or a bug. Besides bug hunting, tracing can be useful to understand a program’s control and data flow. There are many different ways to generate execution traces.

## Tracing with debuggers

Many debuggers offer the ability to trace a program’s execution. Debuggers such as GDB, OllyDbg, and IDA Pro have the ability to generate execution traces along with debugging. Tracing with debuggers is often slower because of every command and instructions have to be processed by the debugger.

## Tracing with system tracers

Strace, Dtrace, LTTng, and SystemTap are examples of system tracers. These system tracers are embedded inside the operating system. They offer good ways to monitor system calls and signals by the program. The downside of using system tracers is that they do not offer much flexible in tracing. The tracing is restricted to system calls and signals. A user cannot define arbitrary points in a program to trace.

## Tracing with instrumentation tools

Pintool, DynamoRio, and Valgrind are binary instrumentation frameworks that can also be used to generate execution traces. These tools perform program analysis in runtime. Custom plugins can be leveraged to monitor and generate program execution. Depending on the configuration of these tools, the performance of these tracers are often much better than using debuggers for tracing. The big advantage of using a debugger is having the disassembly view available simultaneously while determining points for tracing.

## Tracing with TREE

The TREE Tracer plug-in for IDA Pro is built around the IDA Pro debugging framework. Although the performance of our tracer is slower than tracing with instrumentation tools, because of the debugging framework, we believe the flexibility, portability, and power of IDA Pro outweighs its performance disadvantage. We also offer Pintool integration with our TREE Tracer plug-in to take advantage of both speed and power of IDA and Pintool.

The main advantage of using this architecture, besides running within the most popular dissembler, is the ability to leverage IDA Pro’s vast debugging features. With IDA Pro, the user can debug local or remote programs executing in kernel mode, user mode, on Android, Windows, Linux, and 32 bit or 64 bit mode. IDA Pro runs on Windows, Linux, and MacOSX.

For a complete list of IDA Pro debuggers please visit the Hex-Rays website.

https://www.hex-rays.com/products/ida/debugger/

### TREE Tracer Plugin

The TREE Tracer plugin was developed using IDAPython and PySide. One of the main design goals of the TREE Tracer GUI was for IDA Pro users to feel comfortable using TREE. The GUI allows the user to interface either with the IDA Debugger or Pintool. Figure 2 shows the TREE Tracer GUI. The TREE Tracer GUI is very similar to the IDA Debugger process options Figure 3, so if the user is familiar with IDA, he/she will feel comfortable with the TREE Tracer GUI. The user can change and save the current configuration. Tracing can be done either by clicking the run button to start the program from the TREE Tracer or by attaching to an existing program by clicking the attach button.

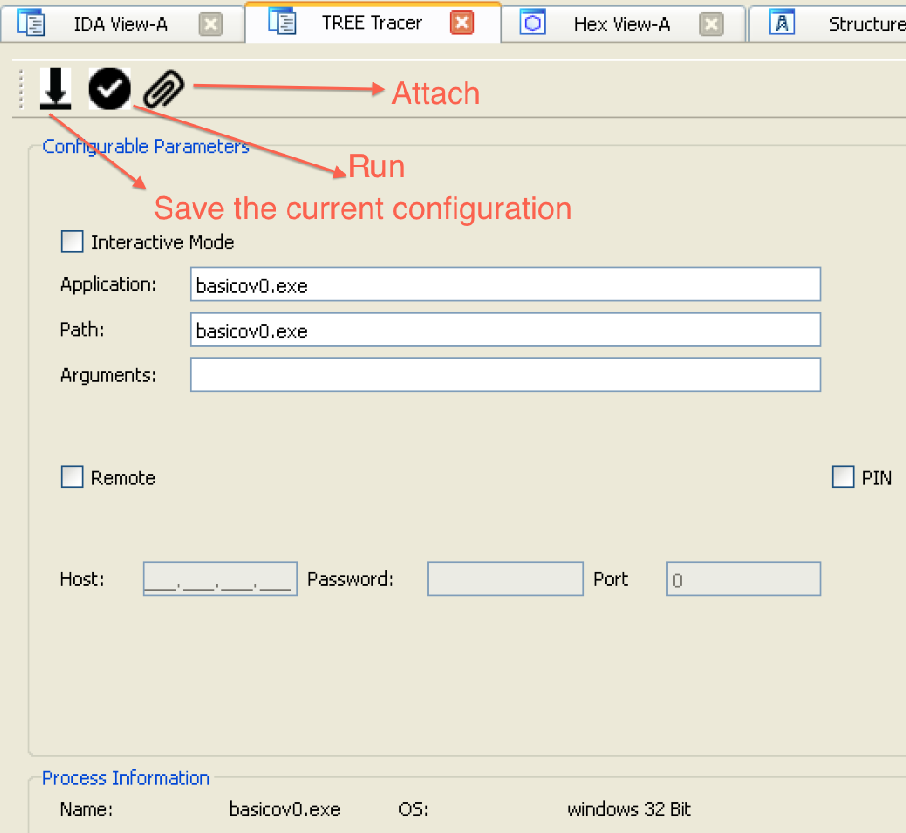


Figure 2 - TREE Tracer GUI

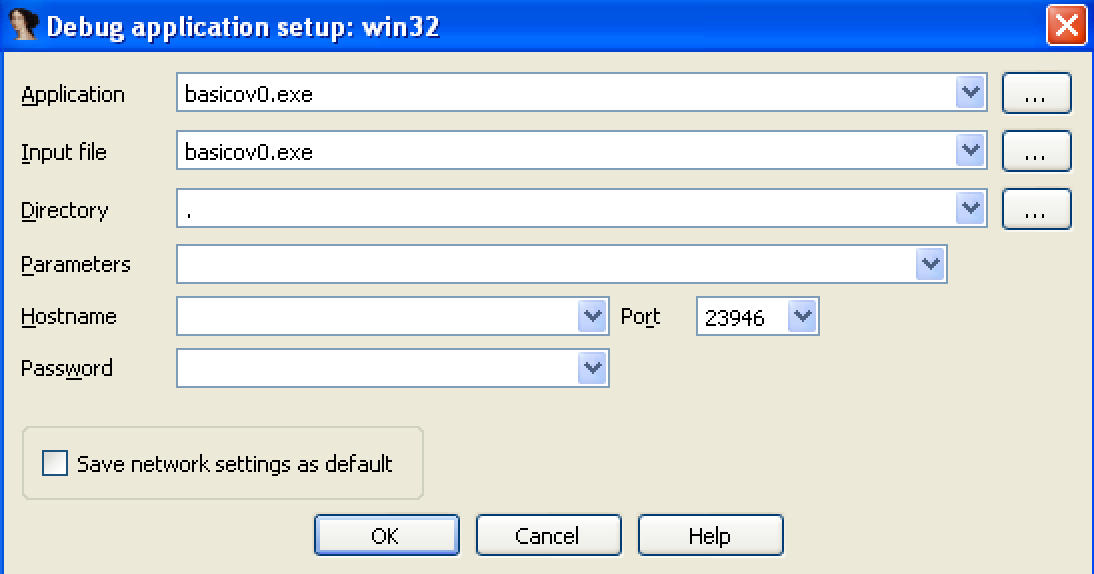


Figure 3 - IDA Pro Debugger Process Options

### Saving configuration settings

With the TREE Tracer GUI, the user can change and save the configuration settings. The configuration settings are stored in “C:\Documents and Settings\[current user]\My Documents\TREE\config.xml” for Windows XP and “C:\Users\[current user]\Documents\TREE\config.xml” for Windows 7.

### Run

With this feature, the TREE Tracer will start the process.

### Attach

With this feature, the TREE Tracer will attach to a running process. The process attach feature only works in interactive mode. Please see [Interactive Mode](#_Interactive_Mode)

## Trace Generation with the TREE Tracer

The IDA Tracer comes with an intuitive GUI to guide the user through trace generation. The user has a choice to use either integrated PinAgent or the IDA Debugger to generate a trace.

### Using the PinAgent

The TREE Tracer GUI communicates with an active PinAgent to generate the trace.

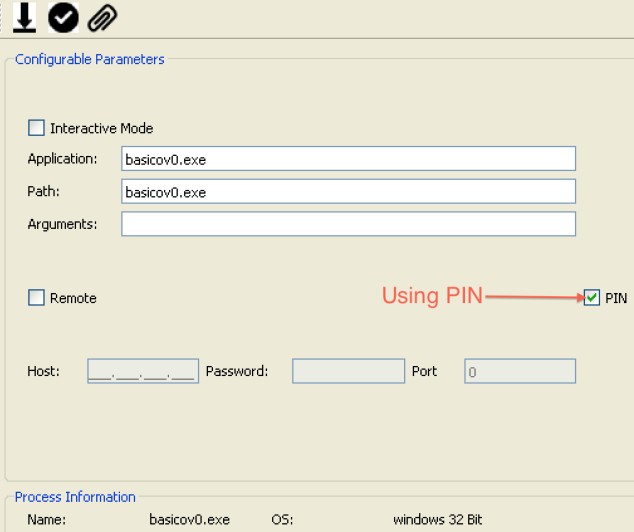


Figure 4 - TREE Tracer with PinAgent

* *This feature is still being developed. We hope to have it ready for our next release.*

### Using the IDA Debugger

The TREE Tracer can run in either interactive or non-interactive mode. Interactive mode allows the user freedom to trace any portion of the disassembly. Non-interactive mode uses filters and API monitoring to generate traces.

### Interactive Mode

In this mode, the user clicks on the “Interactive Mode” check box.

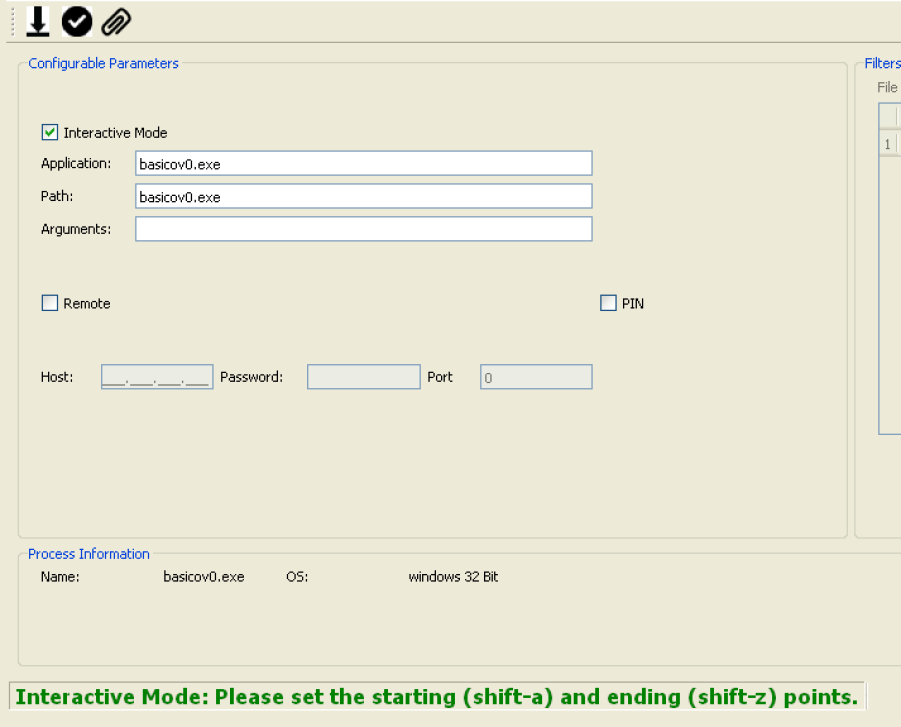


Figure 5 - Interactive Mode

Next, the user selects the starting point to trace by placing the cursor at an address in the IDA View disassembly Window and holding down both the shift and a key. This is where the TREE Trace will start tracing. In this example, it is the address at “.text:00401023 push 3”.

Next, the user selects the ending point to the trace by placing the cursor at an address in the IDA View disassembly Window and holding down both the shift and z key. This is where the TREE Trace will stop tracing. In this example, it is the address at “.text:0040106A lea edx, [ebp+Buffer]”

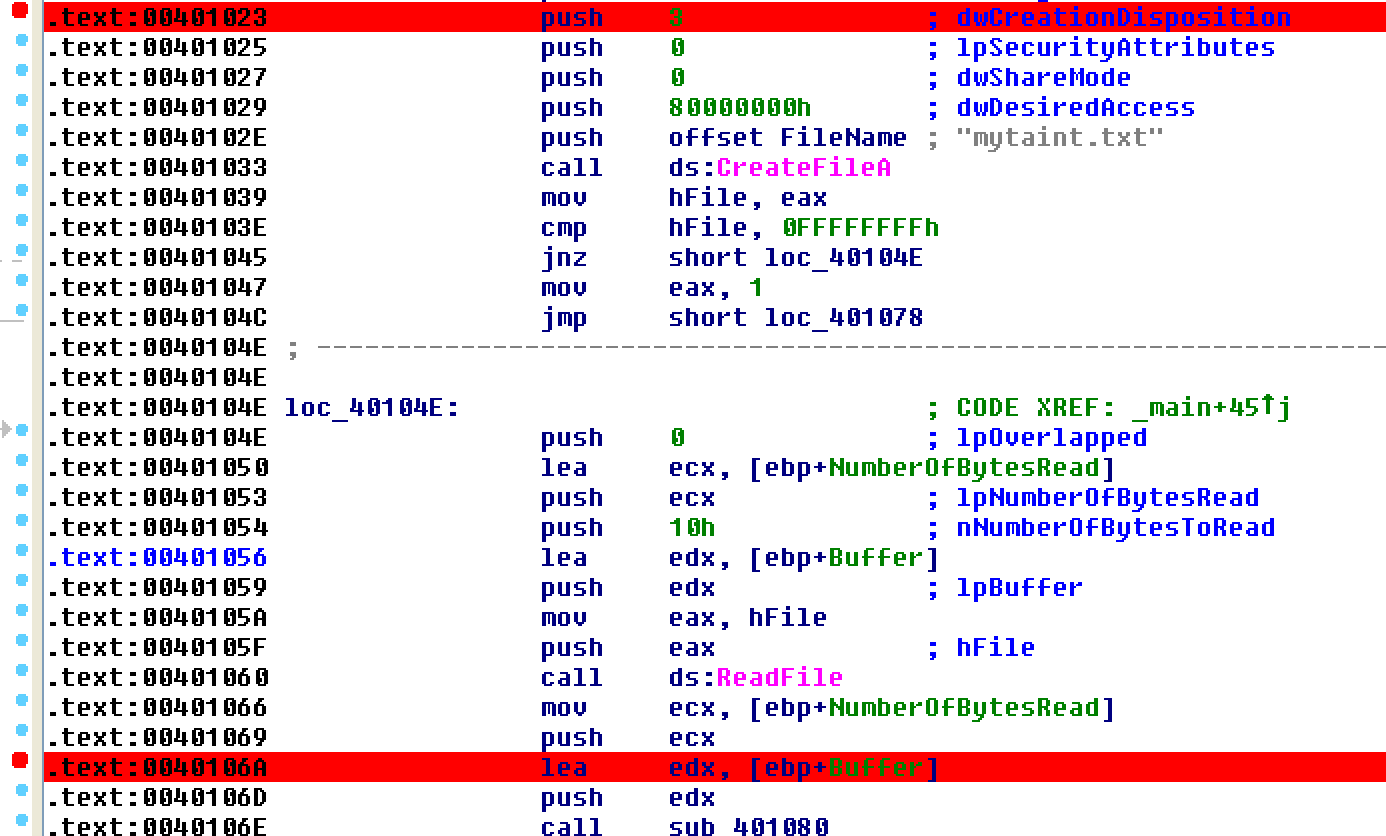


Figure 6 - IDA Pro's Disassembly View

#### Non-Interactive Mode

In this mode, the TREE Tracer handles when to start and stop tracing. The TREE Tracer monitors when a file or network port is open and read from. The user can set up file and/or network filters to control which file or network port to trace.

In the example below, the TREE Tracer will start the program called “basicov.exe”. The trace will start only if basicov.exe opens a file called “mytaint.txt”. The actual tracing will start when basicov.exe reads from mytaint.txt and stop when basicov.exe exits or crashes.

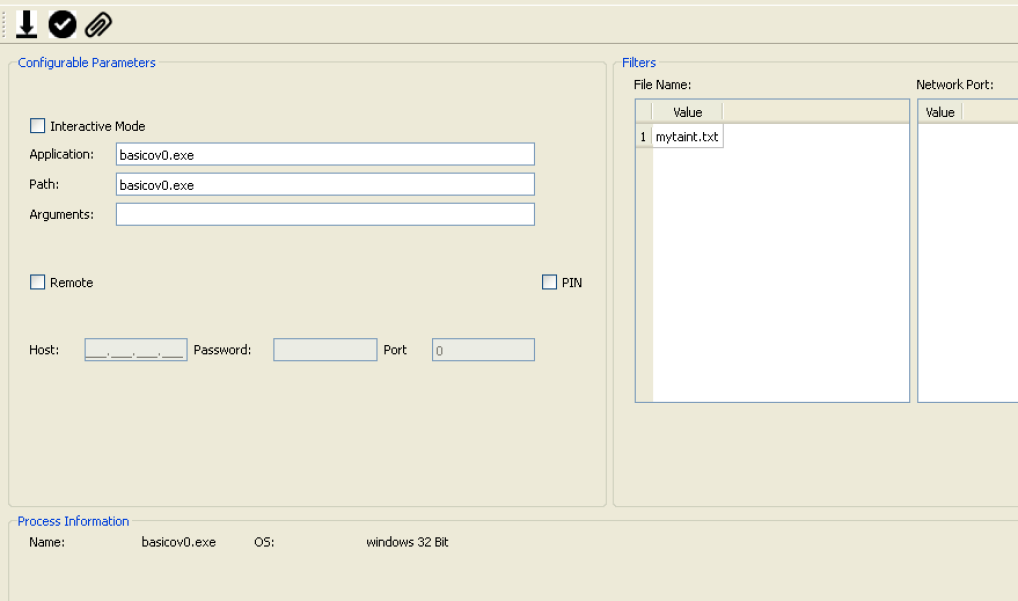


Figure 7 Non-Interactive Mode

## IDA Pro’s Debugging framework

IDA Pro provides the DBG\_Hook API to communicate with the IDA Pro Debugger. The IDA Pro debugger will relay all of its debugging information back to TREE Tracer callback functions.

IDA Debugger

Debug Hook

TREE Callbacks

# 

Figure 8 - IDA Pro's Debugging Framework

# 

Figure 9 - Debug Hook, IDAPython debug hook example

# TREE Analyzer

## Overview of Taint Analysis

Simply speaking, taint analysis is to track information flow inside a program, during a program execution. Information can be anything, from function or system call parameter to a register, or a memory location. Among them, input is one of the most interesting information. So we will use input (taint) in our example in this document.

## Main steps of Taint Analysis at Binary Level

Taint analysis can work at source code, interpreter or binary level but binary level taint analysis tool fits security applications particularly well since all programs run in machine code eventually and binary executable includes code not included in source code. There are basically three steps involved in a complete taint analysis cycle: taint marking, taint tracking and taint checking. The three steps are shown in Figure 1:

Taint Marking

Taint Tracking

Taint Checking

## 

Figure 10 – Complete Taint analysis cycle

## Taint Analysis in TREE

Taint analysis can be done either online or offline. Online taint analysis marks initial taint (*taint source*), tracks taint and checks taint at specific point (*taint sink*) all at the same time the target program executes; offline taint analysis works on an *execution trace* that captures the program states from the previous execution of the target program. TREE uses offline taint analysis because it fits better into the overall TREE system design, Figure 1.

## Trace Parsing and Initial Taint Source Marking

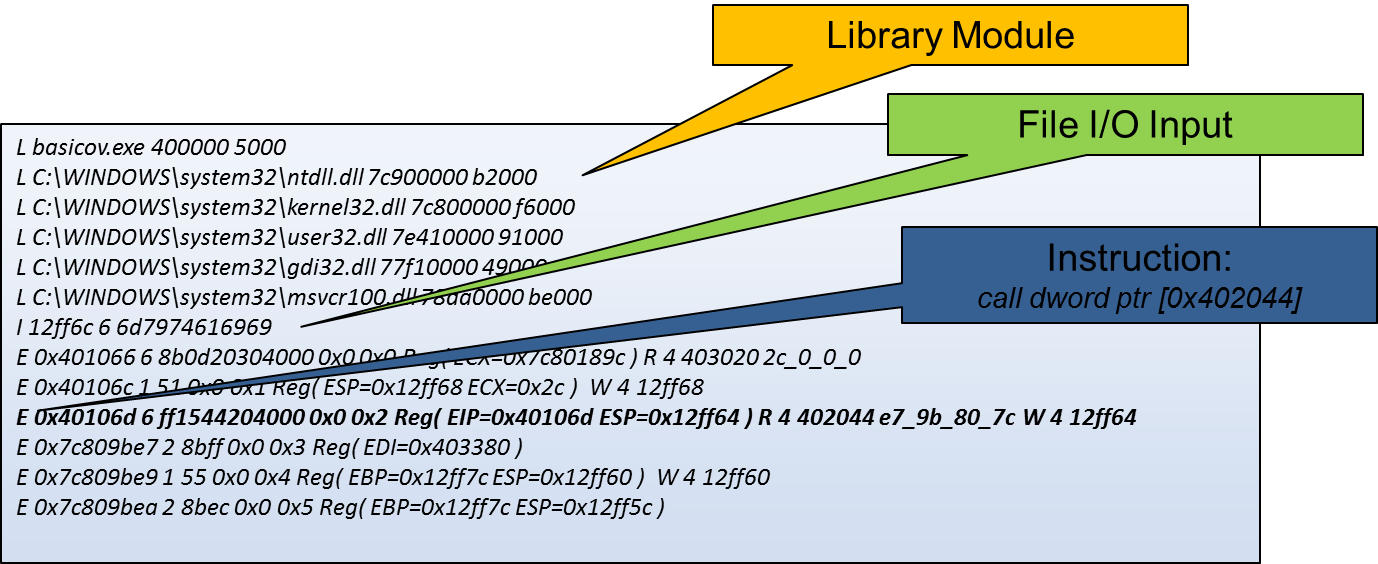


Figure 11 – Sample execution trace output

TREE Trace captures most of the program states:

* Captures a snapshot of the program state at the beginning
* Tracks all instruction level state delta(not the whole state)
* Tracks only relevant read/write memory access (address and value)
* Tracks only relevant register changes and values
* Trace generation is fully automated, no user involvement needed
* PIN Trace records Input/Output buffers for System Call

Initial taint source can be marked through Replayer GUI; otherwise user input recorded in the trace is used as initial taint.

## Taint Tracking

TREE tracks taint through a combination of static taint template/category and instruction-specific propagation.

The static taint template and category is generated using the x86Decoder (based on XED) we developed.

### X86 instruction static taint template

All Intel Architecture instructions are encoded using subsets of the general machine instruction format Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

Inputs to the x86 decoder:

-- Instruction length

-- Instruction encoded bytes

Output:

-- A structure to describe the taint relation between instruction source, destination operands, including explicit and implicit operands. The structure also contains each operand’s type (immediate, register or memory), width in bits, read/write attributes.

*An example:*

Inst\_category=4, Disassembly: sub $0xc, %esp

**src\_operand\_num=2:**

width=32, rw=1, type=2, ea\_string=ESP

width=8, rw=2, type=1, ea\_string=c

**dest\_operand\_num=2:**

width=32, rw=1, type=2, ea\_string=ESP

width=32, rw=3, type=2, ea\_string=EFLAGS[of sf zf af pf cf ]

*Another example:*

*Inst\_category=4, Disassembly: addl -0xc(%ebp), %eax*

***src\_operand\_num=2:***

*width=32, rw=1, type=2, ea\_string=EAX*

*width=32, rw=2, type=3, ea\_string=SEG=SS:BASE=EBP:DISP=-12*

***dest\_operand\_num=2:***

*width=32, rw=1, type=2, ea\_string=EAX*

*width=32, rw=3, type=2, ea\_string=EFLAGS[of sf zf af pf cf ]*

* *eax<- eax, mem[ebp-0xc]*

*EFLAGS[of sf zf af pf cf ] <- eax, mem[*X86 Instruction Categorization and Taint by Category

Of all the x86 instructions, we can roughly classify them into 40 categories; of the 40 categories, we focus on the following most commonly used categories:

* DATAXFER:mov, movzxw,movb,
* BINARY: cmp, sub, add, inc, dec,
* STRINGOP: rep movsdl, rep stosdl,
* COND\_BR: jnz, jl, jz
* LOGICAL: xor, test, or, and, not,
* SHIFT: shr, shl,
* FLAGOP: cld, std,
* PUSH: push
* POP: pop
* RET: ret
* CALL: call
* MISC: leavel, lea,

### Taint Precision: Taint by byte

Taint Relations:

1 To 1:

* + mov %eax,%esi, esi <- eax,
  + push esi, mem[esp] <- esi
  + pop ebp ebp <-mem[esp]

M to N:

* add %ecx, %esi: esi < esi, ecx
* *addl -0xc(%ebp), %eax:*
* *ebp-0xc]*

### Taint Correctness: Over-Taint and Under-Taint

* **Over-taint: Something got tainted when it should not be:**
* xor eax, eax: eax <-0 regardless if eax was tainted or not
* and 0x0, eax : eax <-0 regardless if eax was tainted or not

**Detaint when necessary**

* **Under-taint: Something is not tainted when it should be**
* if (x != 0) y = 1 else y = 0; y is dependent on x, but y doesn’t directly affected by x

**Under-taint is usually because of control-flow dependency, which we can ignore for now**

## Taint Checking and Security Analysis

*This feature is still in development. We will provide more details later.*

# TREE Taint Analysis Testing Framework

## Overview of the Testing Framework:

The framework, shown in Figure 1, has four main steps:

1. Use Standard ReadFile to read input and mark them as initial taints
2. Develop Transformation Functions to transform these inputs
3. Develop Oracles for the Transformation Functions developed in Step 2
4. Compare TA analysis results with Oracles to see if they match; if not, why?

Figure 10 Taint Analysis Testing Framework

1. If there is a bug in Taint Track, fix it and go back to step 1.

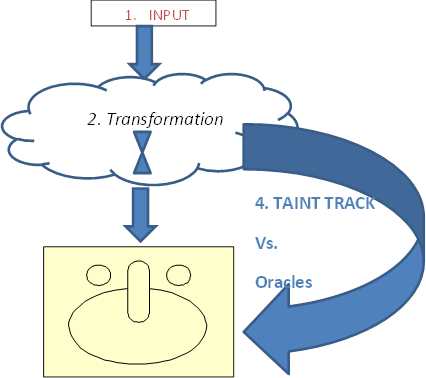


Figure 12– Transformation diagram

## 

## A working template: BasicOV

int main(int argc, char\* argv[])

{

char sBigBuf[16]={0};

hFile = CreateFile("mytaint.txt", // Open One.txt

GENERIC\_READ, // Open for reading

0, // Do not share

NULL, // No security

OPEN\_EXISTING, // Existing file only

FILE\_ATTRIBUTE\_NORMAL, // Normal file

NULL); // No template file

if (hFile == INVALID\_HANDLE\_VALUE)

{

return 1;

}

DWORD dwBytesRead;

ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

Transformation Function is EMPTY

StackOVflow(sBigBuf,dwBytesRead);// Effect

return 0;

}

void StackOVflow(char \*sBig,int num)

{

char sBuf[8]= {0};

for(int i=0;i<num;i++)

{

sBuf[i] = sBig[i];

}

return;

}

***Oracle: derived from your design or debugging***

***Input 1-8 bytes goes to sBuf, 9-12 bytes to overwrite EBP, 13-16 bytes to overwrite EIP***

***Taint Track Result: Taint Graph***

[53]reg\_eip\_0\_40032[0xd5:40032]<-retl {D}42 [42]mem\_0x3afc8c[0xa7:40032]<-movb %dl, -0x8(%ebp,%ecx,1){D}41 [41]reg\_edx\_0\_40032[0xa6:40032][0xac:40032]<-movb (%eax), %dl{D}13

[13]mem\_0x3afca8[-0x1:-1]

[54]reg\_eip\_1\_40032[0xd5:40032]<-retl {D}44

[44]mem\_0x3afc8d[0xb3:40032]<-movb %dl, -0x8(%ebp,%ecx,1){D}43

[43]reg\_edx\_0\_40032[0xb2:40032][0xb8:40032]<-movb (%eax), %dl{D}14

[14]mem\_0x3afca9[-0x1:-1]

[55]reg\_eip\_2\_40032[0xd5:40032]<-retl {D}46

[46]mem\_0x3afc8e[0xbf:40032]<-movb %dl, -0x8(%ebp,%ecx,1){D}45

[45]reg\_edx\_0\_40032[0xbe:40032][0xc4:40032]<-movb (%eax), %dl{D}15

[15]mem\_0x3afcaa[-0x1:-1]

[56]reg\_eip\_3\_40032[0xd5:40032]<-retl {D}48

[48]mem\_0x3afc8f[0xcb:40032]<-movb %dl, -0x8(%ebp,%ecx,1){D}47

[47]reg\_edx\_0\_40032[0xca:40032][0xd0:40032]<-movb (%eax), %dl{D}16

[16]mem\_0x3afcab[-0x1:-1]

## Transformation function and the Oracle:

Some suggestions:

* 1. Compile transformation function in different compiler settings/optimizations
  2. Use different compilers, like gcc, and different optimization settings to compile

A variant of the basicOV program:

ReadFile(hFile, sBigBuf, 16, &dwBytesRead, NULL);

CloseHandle(hFile);

**for(int i=0; i< (dwBytesRead-2); i++)**

**sBigBuf[i] +=sBigBuf[i+1];**

StackOVflow(sBigBuf,dwBytesRead);

## X86 Instruction Coverage and the Priority List

* DATAXFER:mov, movzxw,movb,
* BINARY: cmp, sub, add, inc, dec,
* STRINGOP: rep movsdl, rep stosdl,
* COND\_BR: jnz, jl, jz
* LOGICAL: xor, test, or, and, not,
* SHIFT: shr, shl,
* FLAGOP: cld, std,
* PUSH: push
* POP: pop
* RET: ret
* CALL: call
* MISC: leavel, lea,

# Test Suite:

## Code snippet (transformation-function like) unit level test suites for above instruction categories:

* 10 transformation functions for each category
* 10 transformation functions cover the variants of the instructions
* Transformation functions to be compiled with different compilers(VC, gcc, llvm) and on Windows/Linux platforms

## Real world **programs**

* Benign program or malware

# TREE Visualizer

## Overview of TREE Visualizer

The visualizer component of TREE is the interface to allow for representation of the trace and taint information in graph format. Through the graph format, the taint relationships will be depicted through edge relationships with each graph node representing a single taint.

## Graph Drawing

There are many options in representing the flow of tainted data - the TREE visualizer specifically utilizes node-link diagrams. The vertices or nodes are represented as boxes and the edges are designated as a directional line indicating the transformation flow from one node state to the next.

For more information on graph drawing theory, reference the graphviz theory page.

<http://graphviz.org/Theory.php>

## Data Design

The taint data is stored within a taint object data structure within memory that couples the taint attributes and edge relationships between taint objects. Further taint objects store edge relation attributes for children nodes, where the attributes pertains to the taint policy.

Currently the taint information is passed through to the visualizer in the form of a uniquely named text file. The text file is line-delimited and generated by the Analyzer component with the taint policy as the primary differentiating factor amongst taint graphs.

## Architecture

The taint data is stored within a taint object data structure within memory that couples the taint attributes and edge relationships between taint objects. Further taint objects store edge relation attributes for children nodes, where the attributes pertains to the taint policy.

There are currently two concurrent graphing systems available in the visualizer, the IDA Pro grapher(WinGrapher) and a QT-based grapher. For the QT-based grapher, the layout algorithms currently rely on Scipy/Numpy in order to determine node and edge placement on a QT GraphScene.

## TREE Visualizer Widget

TREE Visualizer exists as a QT Widget alongside the TREE Analyzer. As a whole, the plugin was developed utilizing IDAPython(Python interface for IDA Pro’s APIs) and PySide(Python interface for PyQT). Figure 13 depicts the TREE Visualizer. The first component of the visualizer renders the taint data from the analyzer into a table to allow for a user to sort based on various characteristics. The table also allows for stronger ease in manually following the flow of taint by highlighting child nodes in the context of a selected node.

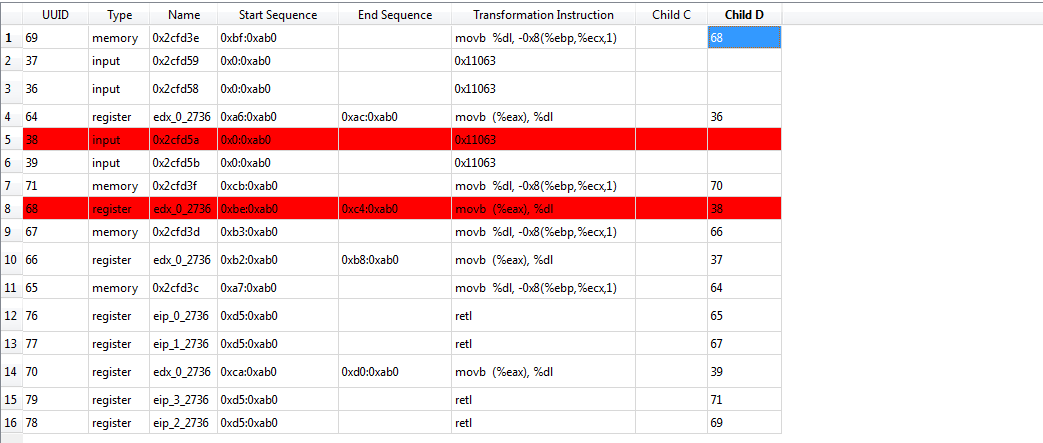


Figure 13 – Taint data table

## Interface

### IDA Pro Grapher

IDA Pro uses WinGraph in generating visual graphs. An example IDA Graph, Figure 14, generated from BasicOV.exe, is depicted in figure 2. Taint nodes, as seen through the IDA Graph are currently uniquely distinguished by their name and color designation for nodes.

* Green - Input taint nodes
* Pink - Register taint nodes
* White - Memory taint nodes
* Red - Sink nodes

The visualizer's other focus point is the taint table. Children nodes are clickable and will show the sequence of taints through contextual highlighting of taint nodes within the table. The highlighting will allow a user to follow the flow of taints within a particular chain.

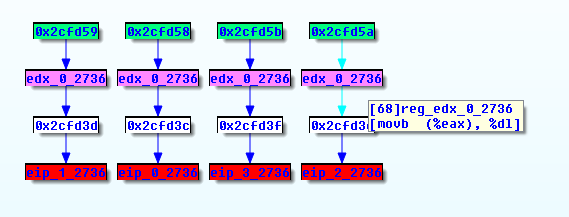


Figure 14 – BasicOV.exe Taint Graph

# References

[1] *Intel Architecture Software Developer's Manual, particularly Volume 2" Instruction Set Reference*

*[2] Eagle, Chris. The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler (2nd Edition). No Starch Press, 2011. Print.*