Next-Generation Debuggers For Reverse Engineering

The ERESI Team
team@eresi-project.org

This Presentation is About...

- The Embedded ERESI debugger: e2dbg
- The Embedded ERESI tracer: etrace
- The ERESI reverse engineering language
- Unification & reconstruction of debug formats
- Program analysis built-ins (focusing on control flow graphs)

The ERESI Project

- Started in 2001 with the ELF shell
- Developed at LSE (EPITA security laboratory)
- Contains more than 10 components
- Featured in 2 articles in Phrack Magazine:
  - The Cerberus ELF Interface (2003)
  - Embedded ELF Debugging (2005)

Limitations of Existing UNIX Debugging Frameworks

- GDB: Use OS-level debugging API (ptrace)
  - Does not work if ptrace is disabled or absent
- Very sensible to variation of the environment (ex: ET_DYN linking of hardened gentoo)
- Strace/Ltrace: use ptrace as well. Very few interaction (command-line parameters)
- None of these frameworks rely on a real reverse engineering language

The ERESI Team

- Started with a single person in 2001 (The ELF shell crew). Remained as it during 3 years.
- Another person joined and developed libasm (disassembling library) since 2002
- A third person developed libdump (the network accessibility library) in 2004-2005
- Since mid-2006: community project (have included up to 10 people)

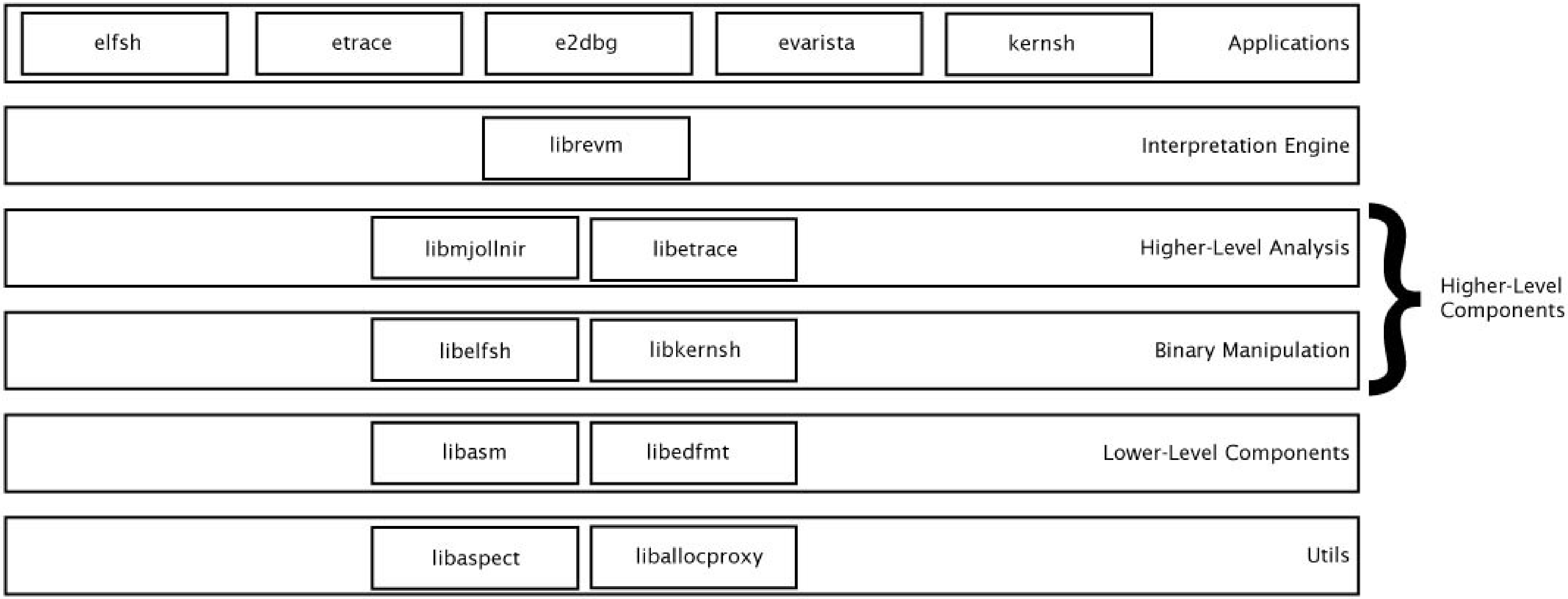
The Modern ERESI Project

- elfsh (and libelfsh): the ELF shell
- e2dbg (and libe2dbg): the embedded ELF debugger
- etrace (and libetrace): the embedded tracer
- kernsh (and libkernsh): code injection and redirection inside the Linux kernel (IA-32 only)
- evarista: a program analyzer written in ERESI

The Modern ERESI Project (2)

- librevm: the language interpreter
- libmjollnir: fingerprinting & graphs library
- libaspect: aspect oriented library (provides many useful data-types)
- libasm: disassembling library with semantic annotations
- libedfmt: the ERESI debug format library
- libui: the user interface (readline-based)

The Modern ERESI Project: Architecture



elfsh etrace e2dbg evarista kernsh — Applications

librevm — Interpretation Engine

libmjollnir libetrace — Higher-Level Analysis

libelfsh libkernsh — Binary Manipulation

Higher-Level Components

libasm libedfmt — Lower-Level Components

libaspect liballocproxy — Utils

ERESI Contributions

- Can debug hardened systems (does not need ptrace)
  - PaX/grsec compatible
- Very effective analysis
  - Improve the performance of fuzzing, heavy-weight debugging
  - No context switching between the debugger and the debuggee – the dbgvm resides in the debuggee

ERESI Contributions (2)

- A reflective framework
  - Possibility to change part of it in runtime without recompilation
- The first real reverse engineering language!!!
  - Hash tables
  - Regular expressions
  - Loops, conditionals, variables
  - The complete ELF format objects accessible from the language

The ERESI Language: Example 1

```
load /usr/bin/ssh

set $entnbr 1.sht[.dynsym].size
div $entnbr 1.sht[.dynsym].entsize
print Third loop until $entnbr :
foreach $idx of 0 until $entnbr
  print Symbol $idx is 1.dynsym[$idx].name
forend

unload /usr/bin/ssh
```

The ERESI Language: Example 2

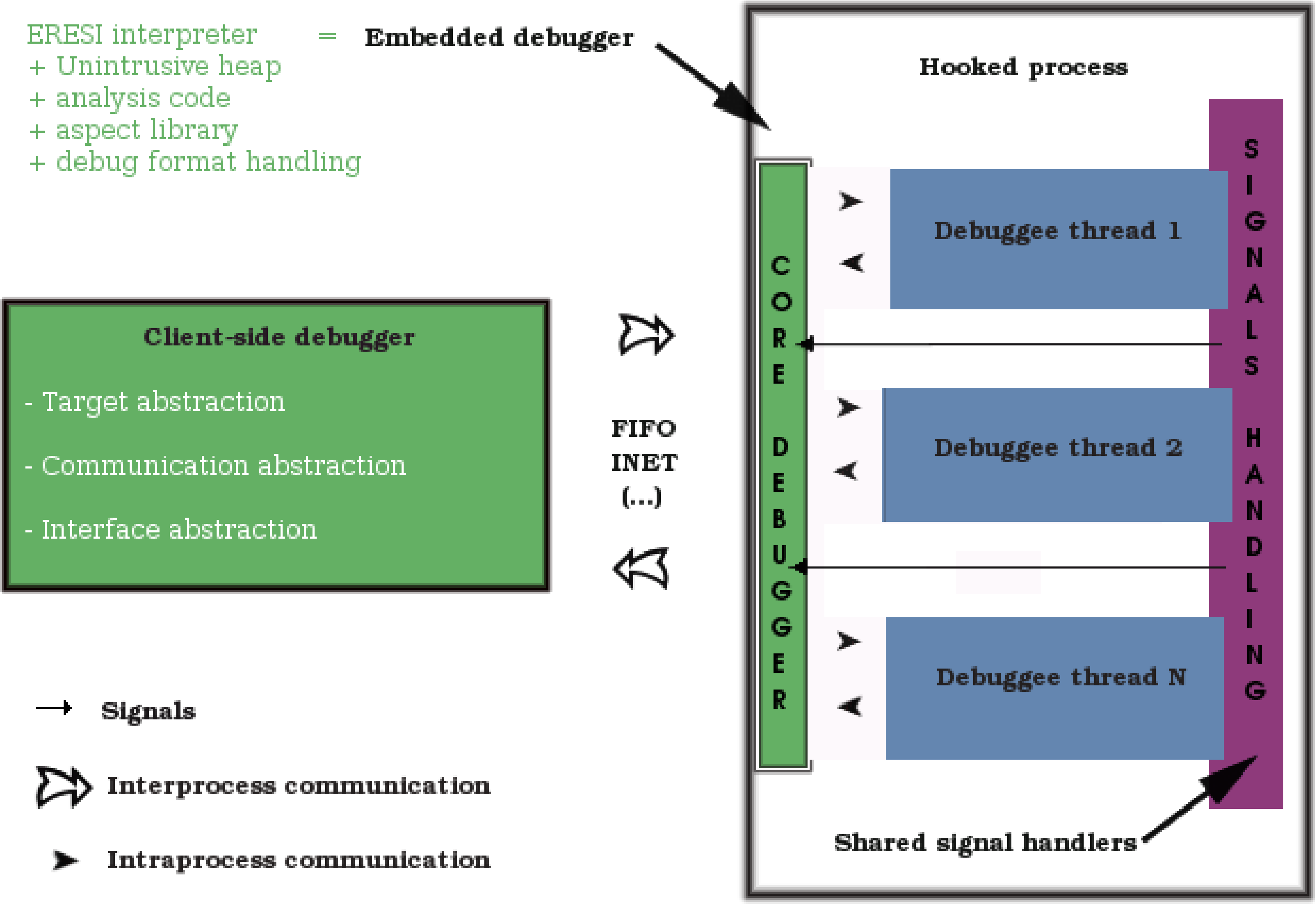
```
add $hash[hname] Intel
add $hash[hname] Alpha
add $hash[hname] Sparc32
add $hash[hname] Mips
add $hash[hname] Sparc64
add $hash[hname] AMD
add $hash[hname] Pa-risc
foreach $elem of hname matching Sparc
      print Regex Matched $elem
 endfor
```

List of Available Hash Tables

- Basic blocks (key: address)
- Functions (key: address)
- Regular expression applied on the key
- Many dozen of hash tables (commands, objects...)
  - See 'tables' command of ERESI
- Currently not supported: hash table of instructions, of data nodes (too many elements) => need of demand-driven analysis

e2dbg, The Embedded ELF Debugger

- Does not use ptrace. Does not have to use any OS level debug API. Evades PaX and grsecurity
- Proof of concept developed on Linux/x86
- Scriptable using the ERESI language
- Support debugging of multithreads
- No need of ANY kernel level code (can execute in hostile environment)



ERESI interpreter = Embedded debugger
+ Unintrusive heap
+ analysis code
+ aspect library
+ debug format handling

Hooked process

CORE DEBUGGER

Debuggee thread 1

Debuggee thread 2

Debuggee thread N

SIGNALS HANDLING

Shared signal handlers

Client-side debugger

- Target abstraction

- Communication abstraction

- Interface abstraction

FIFO
INET
(...)

→ Signals

Interprocess communication

➤ Intraprocess communication

e2dbg: Features

- Classical features:
  - breakpoints (using processor opcode or function redirection)
  - stepping (using sigaction() syscall)
- Allocation proxying
  - keep stack and heap unintrusiveness
  - NOT a memory protection technique
- Support for multithreading

Allocation Proxying

- We manage two different heap allocator in a single process:

```
int hook_malloc(int sz)
{
   if (debugger)
      return (aproxy_malloc(sz));
   return (orig_malloc(sz))
}
```

Debugging Formats

- Describe each element of a program
  - Give names and position of:
    - Variables
    - Functions
    - Files
    - …
  - Store type information

Debugging Formats – Issues

- Distinction of debugging format
  - stabs, dwarf, stabs+, dwarf2, gdb, vms...
  - Different ways to parse, read, store…
- For example with stabs and dwarf2
  - Stabs does not contain any position reference
    - ▯ You store the whole parsing tree
  - Dwarf2 use read pattern apply directly on data
    - ▯ You cannot store everything (too big)
  - …

Unified Debugging Format

- Parsing
  - So we can read the debugging format
- Transforming
  - We transform it to a uniform representation
  - Keep only useful information
- Cleaning
  - We keep only the unified debugging format
- New debugging format
  - We change only backend part
- Register types on ERESI type engine

Embedded ELF Tracer

- Tracer using ERESI framework
- Tracing internal and external calls
- Dynamic and supports multiple architecture
  - It does not use statically stored function prototypes
  - Use gcc to reduce architecture dependence
- Work with and without debugging format
- Recognize string, pointers and value

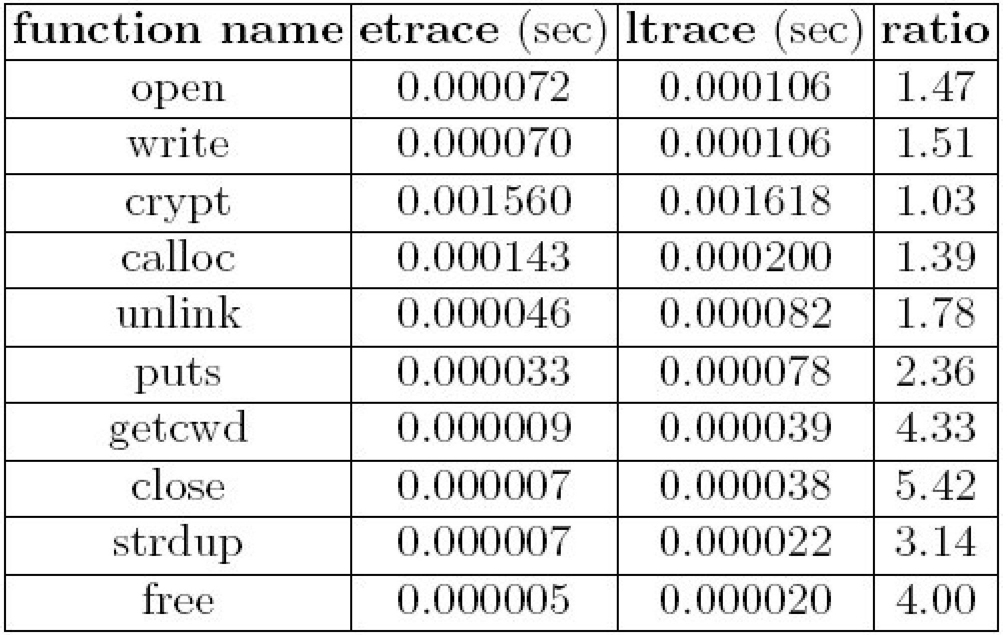
Embedded ELF Tracer – script

```
#!/usr/local/bin/elfsh32
load ./sshd
traces add packet_get_string
traces create privilege_sep
traces add execv privilege_sep
traces create password
traces add auth_password password
traces add sys_auth_passwd password
save sshd2
```

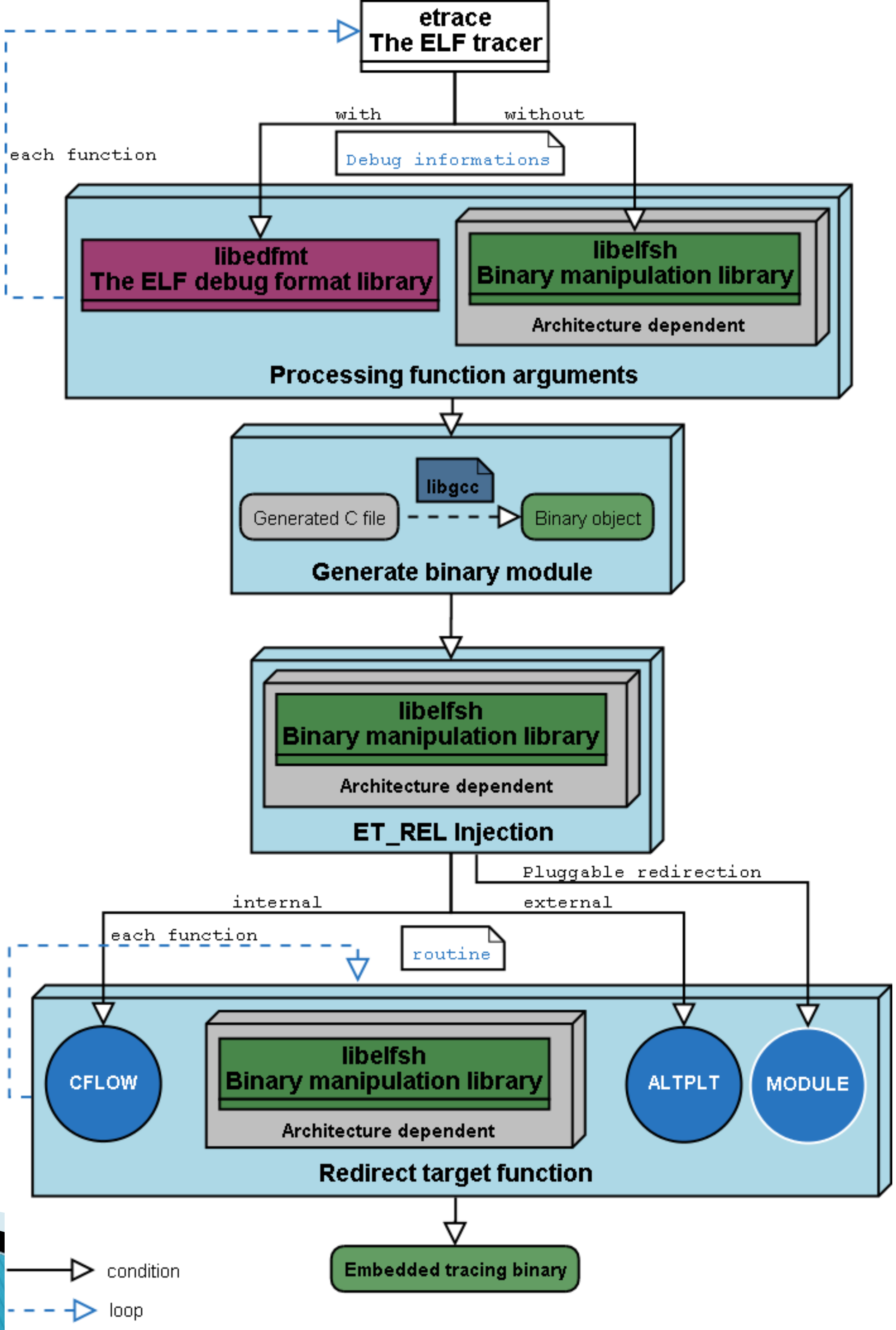
Etrace – Output on sshd

```
+ execv(*0x80a5048 “(…)/openssh-4.5p1/sshd2",
  *0x80aa0a0)
  + packet_get_string(*u_int length_ptr: *0xbf8f4738)
  - packet_get_string = *0x80ab9f0 "mxatone"
debug1: Attempting authentication for mxatone. (…)
  + packet_get_string(*u_int length_ptr: *0xbf8f42fc)
  - packet_get_string = *0x80a9970 "test1"
  + auth_password(*Authctxt authctxt: *0x80aaca0, void*
    password: *0x80b23a8 "test1")
    + sys_auth_passwd(*Authctxt authctxt: *0x80aaca0,
  void*            password: *0x80b23a8 "test1")
    - sys_auth_passwd = 0x0
  - auth_password = 0x0
```

Etrace – Performance



function name	etrace (sec)	ltrace (sec)	ratio
open	0.000072	0.000106	1.47
write	0.000070	0.000106	1.51
crypt	0.001560	0.001618	1.03
calloc	0.000143	0.000200	1.39
unlink	0.000046	0.000082	1.78
puts	0.000033	0.000078	2.36
getcwd	0.000009	0.000039	4.33
close	0.000007	0.000038	5.42
strdup	0.000007	0.000022	3.14
free	0.000005	0.000020	4.00

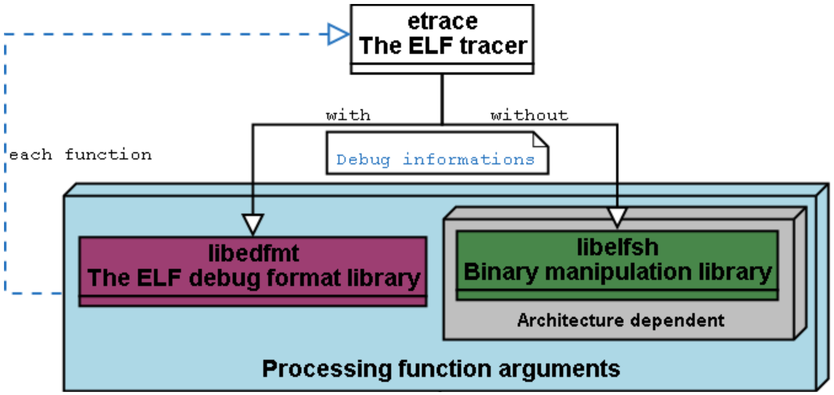

etrace
The ELF tracer
with
without
each function
Debug informations
libedfmt
The ELF debug format library
libelfsh
Binary manipulation library
Architecture dependent
Processing function arguments
libgcc
Generated C file
Binary object
Generate binary module
libelfsh
Binary manipulation library
Architecture dependent
ET_REL Injection
Pluggable redirection
internal
external
each function
routine
CFLOW
libelfsh
Binary manipulation library
Architecture dependent
ALTPLT
MODULE
Redirect target function
Embedded tracing binary
condition
loop

Embedded ELF Tracer

- Trace backend
  - Analyze target functions to determine number of parameters
  - Create proxy functions
- Embedded tracer
  - Inject proxy functions in the binary
  - Redirect calls into our proxy functions
  - Create a new binary
- Automated using the ELF tracer

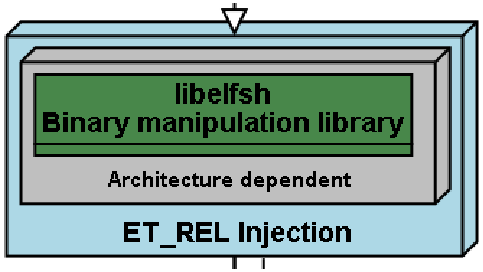
Etrace – Processing Function Arguments

- With debugging information
  - Extract arguments information
    - size
    - names
    - type names
    - …
- With architecture dependent argument counting
  - Backward analysis
  - Forward analysis



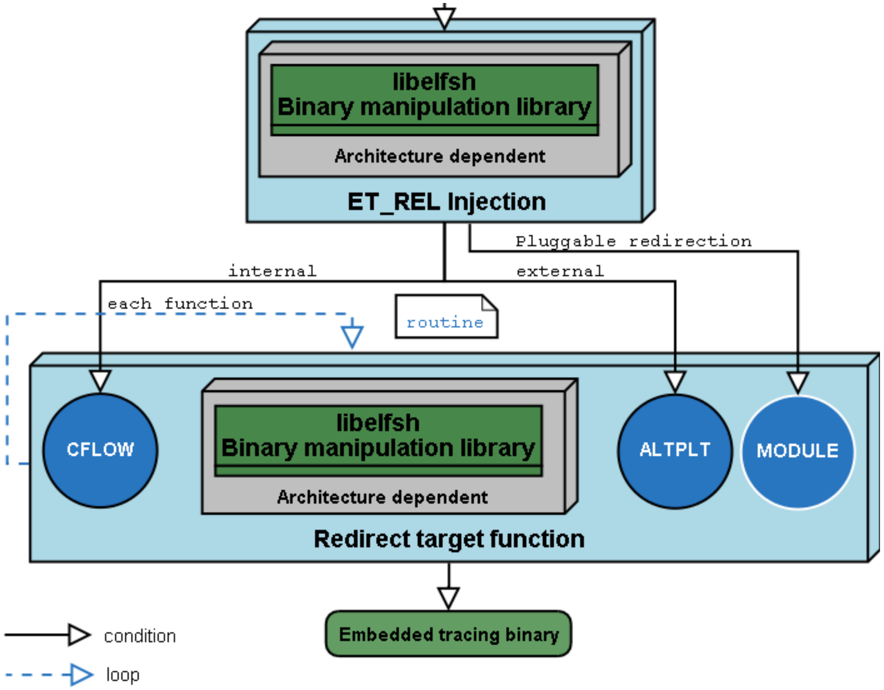
Libelfsh – ET_REL Injection

- ET_REL injection principle
  - Add a binary module directly on target binary
- Merge symbols and sections list
- Section injection
  - Code sections
  - Data sections



Libelfsh – Redirect Target Function

- Internal function
  - CFLOW technique
- External function
  - ALTPLT technique



A Graph Analyzer

- Graph analyzers
    - Identify blocks and functions
    - Identify links (calls and jumps)
    - Build a graph with this info
- Control Flow Graphs (CFGs)
    - Inter-blocks CFGs vs. Interprocedural CFGs
    - Main instrument to Control Flow analysis

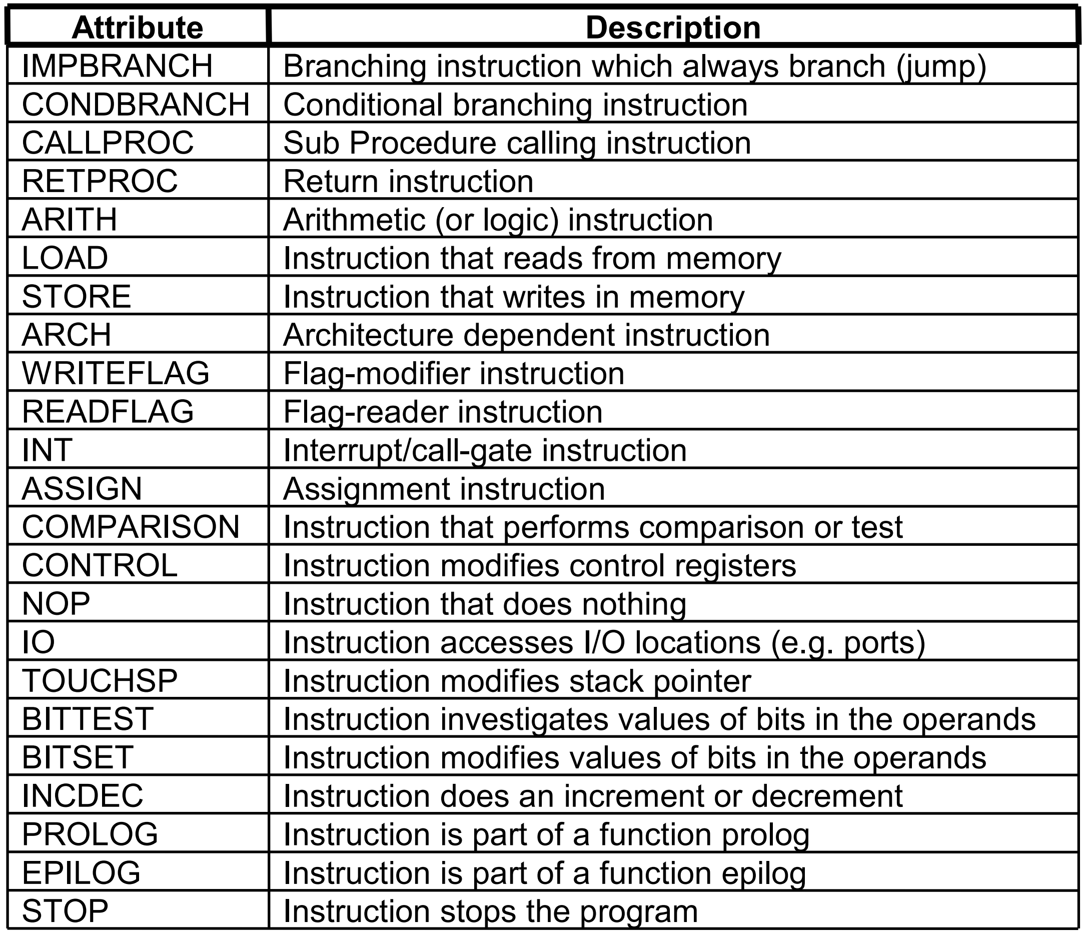
A Graph Analyzer

- Control Flow Analysis
  - Essential to some kinds of further analysis and to optimization
  - Gives information about properties such as
    - Reachability
    - Dominance
    - …

A Graph Analyzer – Libasm

- Libasm
  - Lowest layer of this application
  - Multi-architecture disassembling library
    - Intel IA-32
    - SPARC V9
    - In the near future, MIPS
  - Unified list of semantic attributes

A Graph Analyzer – Libasm



Attribute	Description
IMPBRANCH	Branching instruction which always branch (jump)
CONDBRANCH	Conditional branching instruction
CALLPROC	Sub Procedure calling instruction
RETPROC	Return instruction
ARITH	Arithmetic (or logic) instruction
LOAD	Instruction that reads from memory
STORE	Instruction that writes in memory
ARCH	Architecture dependent instruction
WRITEFLAG	Flag-modifier instruction
READFLAG	Flag-reader instruction
INT	Interrupt/call-gate instruction
ASSIGN	Assignment instruction
COMPARISON	Instruction that performs comparison or test
CONTROL	Instruction modifies control registers
NOP	Instruction that does nothing
IO	Instruction accesses I/O locations (e.g. ports)
TOUCHSP	Instruction modifies stack pointer
BITTEST	Instruction investigates values of bits in the operands
BITSET	Instruction modifies values of bits in the operands
INCDEC	Instruction does an increment or decrement
PROLOG	Instruction is part of a function prolog
EPILOG	Instruction is part of a function epilog
STOP	Instruction stops the program

A Graph Analyzer – Libasm

- The instruction semantic annotations
  - Works with non-mutually exclusive ‘types’
  - Provides means to ‘blindly’ analyze an instruction
  - eg. Control Flow analysis!

A Graph Analyzer – Libasm

- Libasm vectors
  - Storage of pointers to opcode handling functions
  - Runtime dumping and replacing of vectors
    - Built-in language constructs
    - Easy-made opcode tracer!

A Graph Analyzer – libmjollnir

- Libmjollnir
  - Upper-layer component
  - Code fingerprinting and program analysis
- CFG construction
  - Libmjollnir treats both: blocks and functions
  - Separate representations (structures)

A Graph Analyzer – libmjollnir

- Containers
  - Generic structures to encapsulate blocks and functions
  - Have linking (input and output links) information
  - Have a pointer to data and type information to interpret this data accordingly

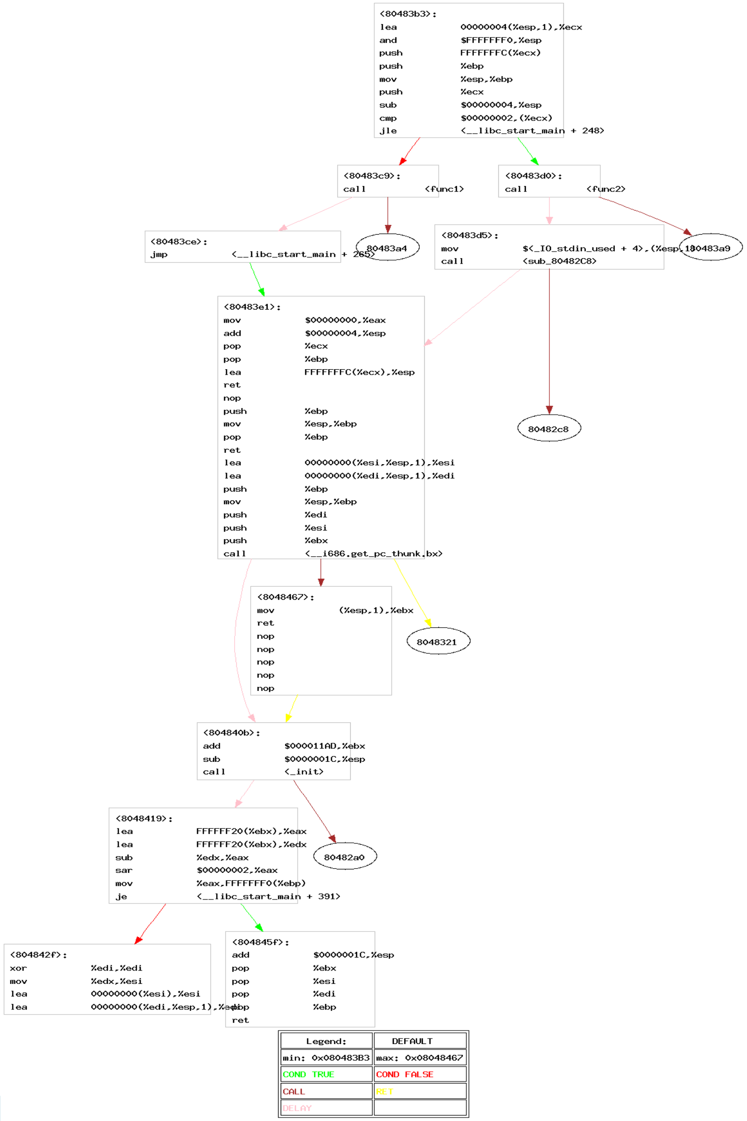
A Graph Analyzer – libmjollnir

- Containers
  - Allow for more abstract graph analysis (analyzing a graph of containers)
  - In the future, may also store data nodes (Data Flow analysis)
  - Also for the future, containers of containers
    - Even higher abstraction of links and relationships

A Graph Analyzer – Example

```
#include <stdio.h>
void func1() {}
void func2() { func1(); }
int main(int argc,char **argv) {
   if (argc > 2) {
      func1();
   }
   else {
      func2();
      printf("hey there!\n");
   }
   return 0;
}
```

A Graph Analyzer – Example



```
<80483b3>:
lea         00000004(%esp,1),%ecx
and         $FFFFFFF0,%esp
push        FFFFFFFC(%ecx)
push        %ebp
mov         %esp,%ebp
push        %ecx
sub         $00000004,%esp
cmp         $00000002,(%ecx)
jle         <__libc_start_main + 248>
```

```
<80483c9>:
call        <func1>
```

```
<80483d0>:
call        <func2>
```

```
<80483ce>:
jmp         <__libc_start_main + 265>
```

```
<80483d5>:
mov         $<_IO_stdin_used + 4>,(%esp,1)
call        <sub_80482C8>
```

```
<80483e1>:
mov         $00000000,%eax
add         $00000004,%esp
pop         %ecx
pop         %ebp
lea         FFFFFFFC(%ecx),%esp
ret
nop
push        %ebp
mov         %esp,%ebp
pop         %ebp
ret
lea         00000000(%esi,%esp,1),%esi
lea         00000000(%edi,%esp,1),%edi
push        %ebp
mov         %esp,%ebp
push        %edi
push        %esi
push        %ebx
call        <__i686.get_pc_thunk.bx>
```

```
<8048467>:
mov         (%esp,1),%ebx
ret
nop
nop
nop
nop
nop
```

```
<804840b>:
add         $000011AD,%ebx
sub         $0000001C,%esp
call        <_init>
```

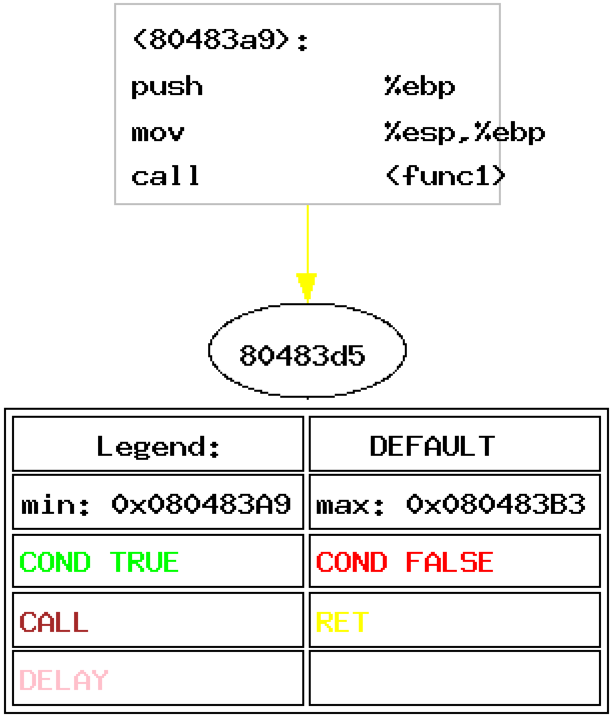
```
<8048419>:
lea         FFFFFF20(%ebx),%eax
lea         FFFFFF20(%ebx),%edx
sub         %edx,%eax
sar         $00000002,%eax
mov         %eax,FFFFFFF0(%ebp)
je          <__libc_start_main + 391>
```

```
<804842f>:
xor         %edi,%edi
mov         %edx,%esi
lea         00000000(%esi),%esi
lea         00000000(%edi,%esp,1),%edi
```

```
<804845f>:
add         $0000001C,%esp
pop         %ebx
pop         %esi
pop         %edi
pop         %ebp
ret
```

Legend:	DEFAULT
min: 0x080483B3	max: 0x08048467
COND TRUE	COND FALSE
CALL	RET
DELAY	

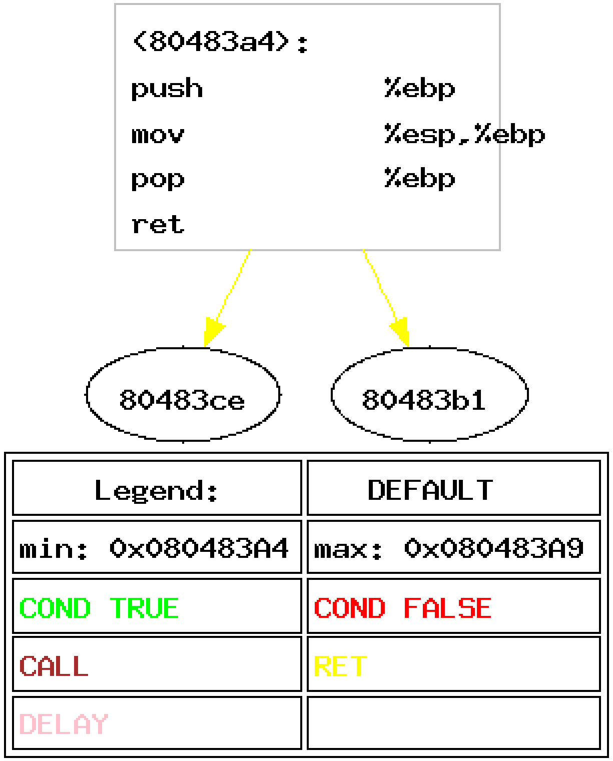
A Graph Analyzer – Example



```
<80483a9>:
push            %ebp
mov             %esp,%ebp
call            <func1>
```

Legend:	DEFAULT
min: 0x080483A9	max: 0x080483B3
COND TRUE	COND FALSE
CALL	RET
DELAY	

A Graph Analyzer – Example



```
<80483a4>:
push          %ebp
mov           %esp,%ebp
pop           %ebp
ret
```

Legend:	DEFAULT
min: 0x080483A4	max: 0x080483A9
COND TRUE	COND FALSE
CALL	RET
DELAY	

Conclusion

- New foundations for reverse engineering and debugging of closed-source software using in-process analysis
- A language approach for reversing
- Many concrete applications (embedded tracer and debugger)

The Near Future

- Binding of demand-driven dataflow analysis in the ERESI language
- Program transformation builtins for custom decompilation
- Kernel debugging and tracing
- More portability (OS/Architectures)
- More integration between the components (tracer/debugger mostly)

Questions ?

- Thank you for your attention
- If you are interested in joining us, come to talk after the conference.
- The source code of the current version (0.8a19) is available at our web CVS:
  - http://cvs.eresi-project.org/
- Also, don’t forget to visit our website:
  - http://www.eresi-project.org/

Next-Generation Debuggers For Reverse Engineering

The ERESI Team
team@eresi-project.org