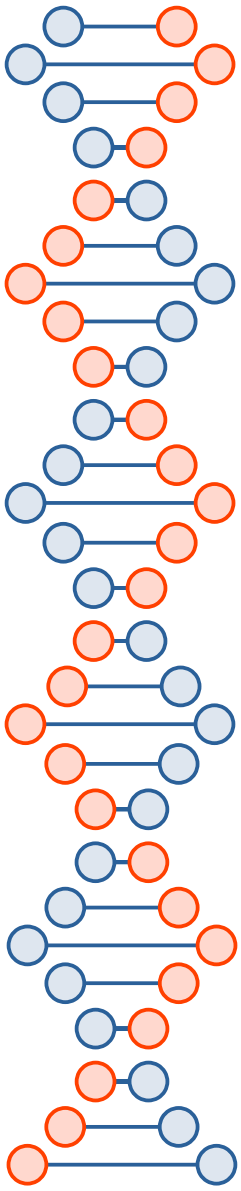
The “Shiva runtime environment”

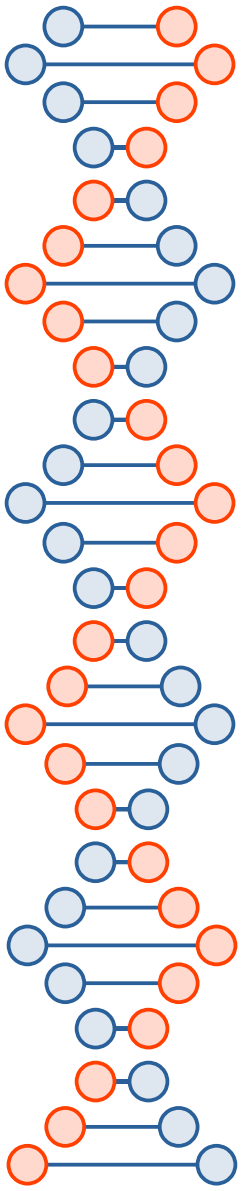
Advancing the programmability of the Linux process address space

Ryan “elfmaster” O’Neill: ryan@bitlackeys.org



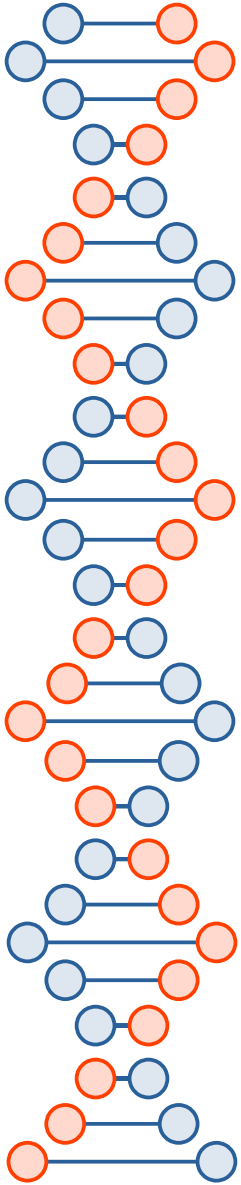
What is Shiva exactly?

- A weird machine: programmable runtime engine
- Custom dynamic linker:
  - Loads ELF micro-programs into the address space
- Shiva provides a rich API for program transformation, tracing, hooking, and debugging.
- Think “LKM’s (Loadable kernel modules) for userland processes”.
- A programmable debugging engine that does ***NOT require the ptrace system call***



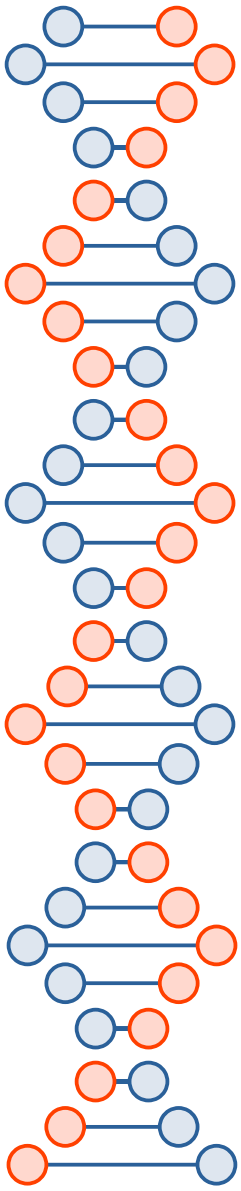
What can Shiva be used for?

- Building debuggers and tracers
- Designing process Security modules
  - Anti-exploitation, Sandboxing, Process hardening
- In-memory fuzzing harnesses
- Software profiling
- Virus detection engines
- Malware unpackers
- Hot-patching code and data
- Userland-Rootkit detection & disinfection
- Extremely fast instrumentation, hooking, and process injection of all types *(Without the use of SYS_ptrace)*.



From a blackhat perspective

- Shiva can be used to crack software
- Design powerful in-process rootkits
- Perform Modular virus infection: “Preloading the linker for fun and profit”: https://tmpout.sh/2/6.html
- Create ELF binary packers



How does Shiva run?

- Shiva loads and links “**Shiva modules**” at runtime
- Shiva loads the module “/opt/shiva/modules” directory

1. Shiva can be invoked directly as an executable
   - *Shiva works as a userland-exec to load and map the target executable into its own process address space (See grugqs work on ul_exec: https://grugq.github.io/docs/ul_exec.txt)*
   - *i.e.* ***`./shiva /bin/test_prog`***

2. Shiva can be invoked indirectly as a runtime interpreter *(i.e. A program declares shiva as the dynamic linker)*
   - *The kernel loads the target executable and loads Shiva as the interpreter*
   - *Good for writing modules that are linked at every single execution*
   - *i.e.* ***`/bin/test_prog`***

Shiva ran directly on program ./test

- Shiva is loading a module: **`pltgot_hook.o`**
- This module hooks the PLT function '**`puts`**' and modifies the string output

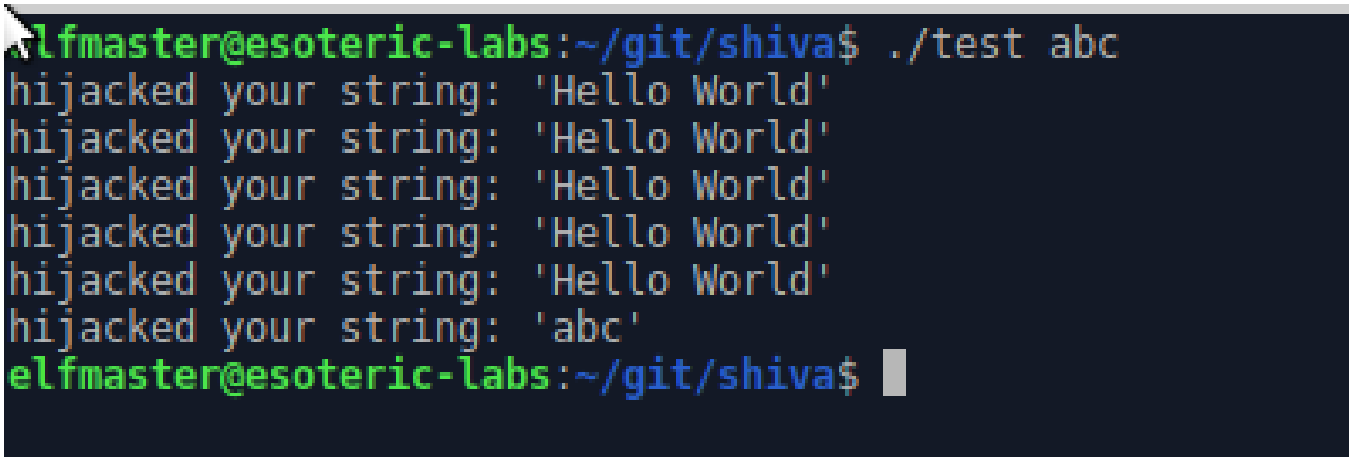
```
elfmaster@esoteric-labs:~/git/shiva$ ./test abc
Hello World
Hello World
Hello World
Hello World
Hello World
abc
elfmaster@esoteric-labs:~/git/shiva$ ./shiva test abc
hijacked your string: 'Hello World'
hijacked your string: 'Hello World'
hijacked your string: 'Hello World'
hijacked your string: 'Hello World'
hijacked your string: 'Hello World'
hijacked your string: 'abc'
elfmaster@esoteric-labs:~/git/shiva$
```

Shiva interpreter path

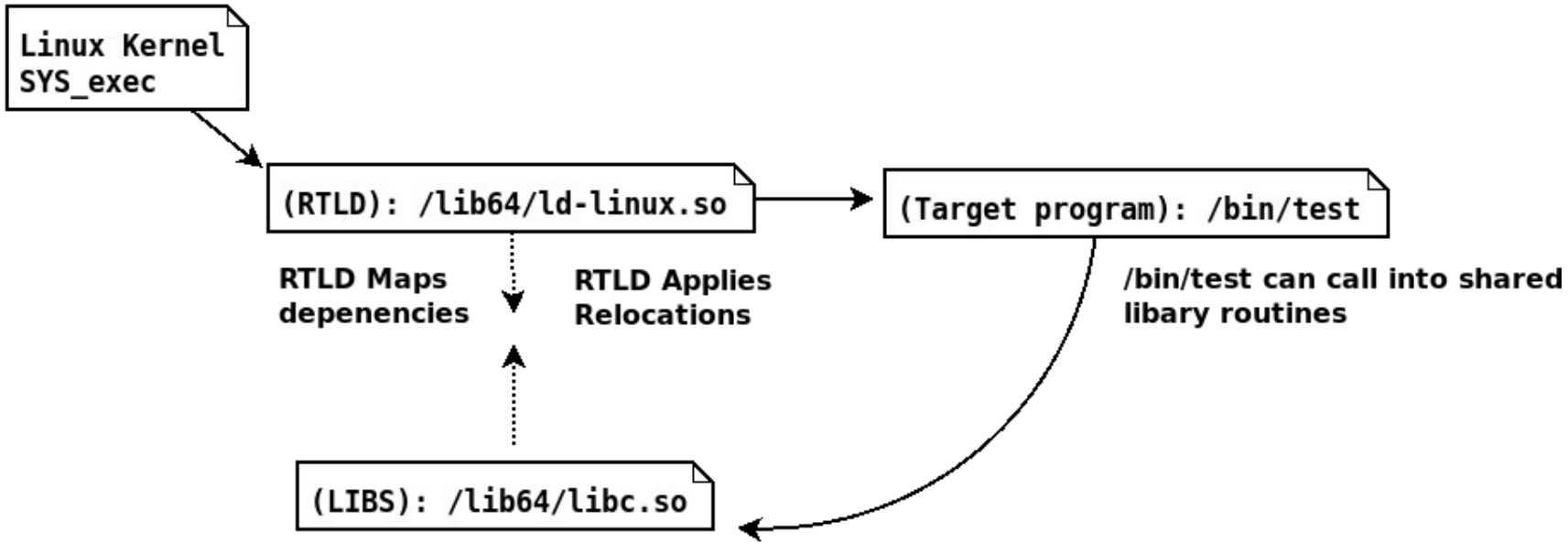
- The INTERP segment contains the path to ‘shiva’ instead of “ld-linux.so”.

```
Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1080
There are 14 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x0000000000000310 0x0000000000000310  R      0x8
  INTERP         0x0000000000000350 0x0000000000000350 0x0000000000000350
                 0x0000000000000020 0x0000000000000020  R      0x1
      [Requesting program interpreter: /home/elfmaster/git/shiva/shiva]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
```

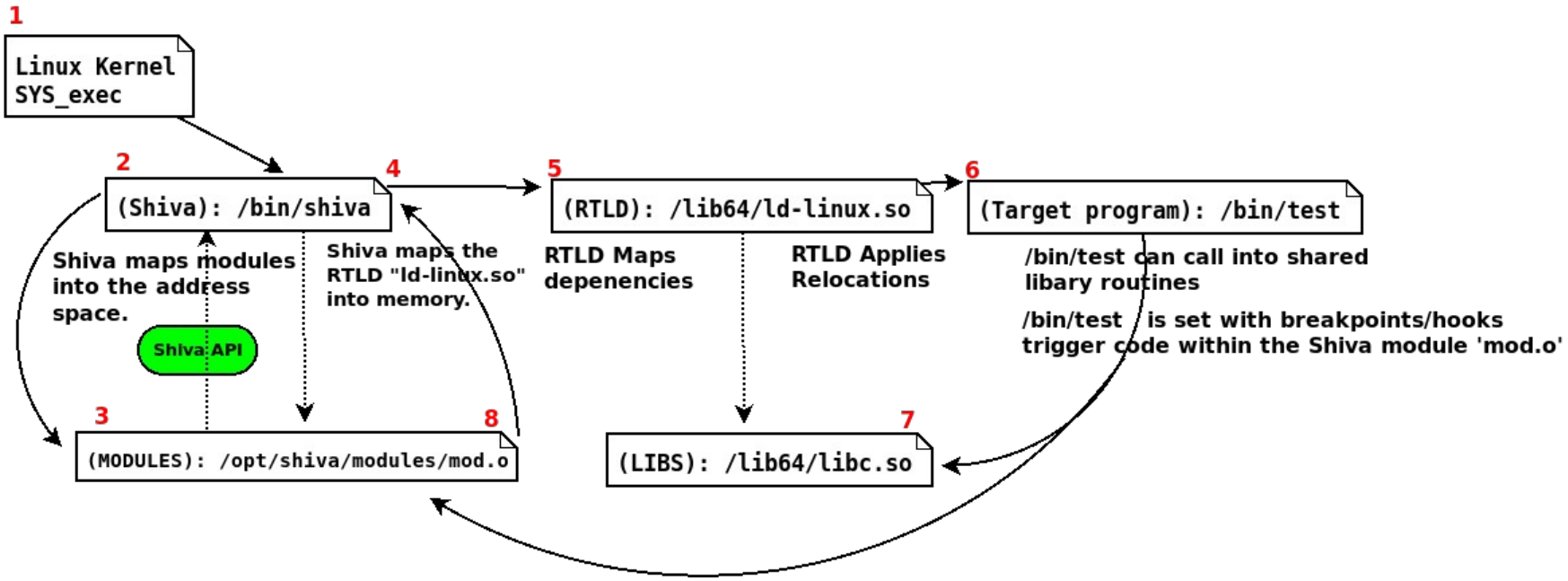
Shiva running as the interpreter for ./test

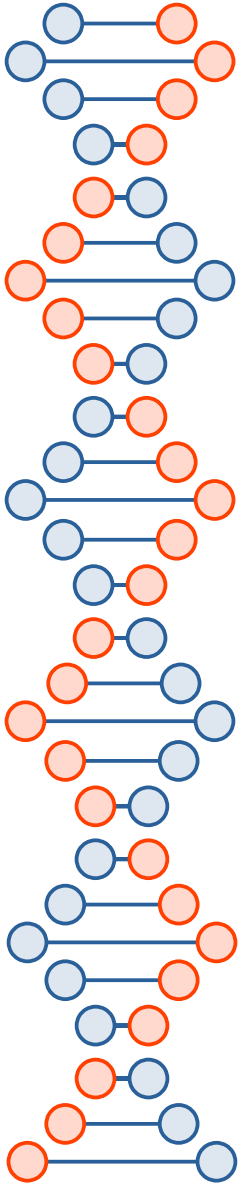


Process execution flow of a normal dynamically linked ELF



Process execution flow with Shiva as an interpreter.

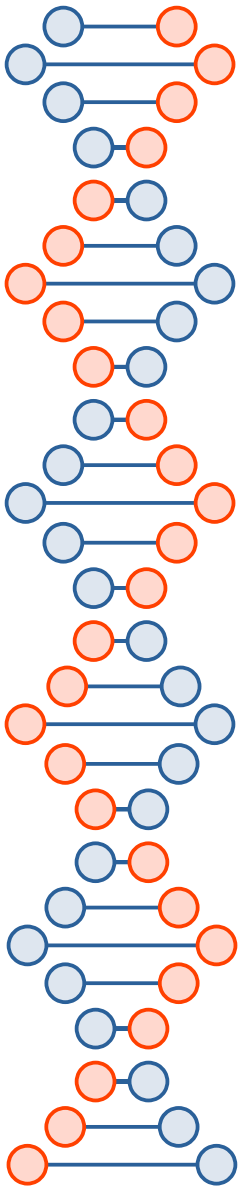




Understanding Shiva modules

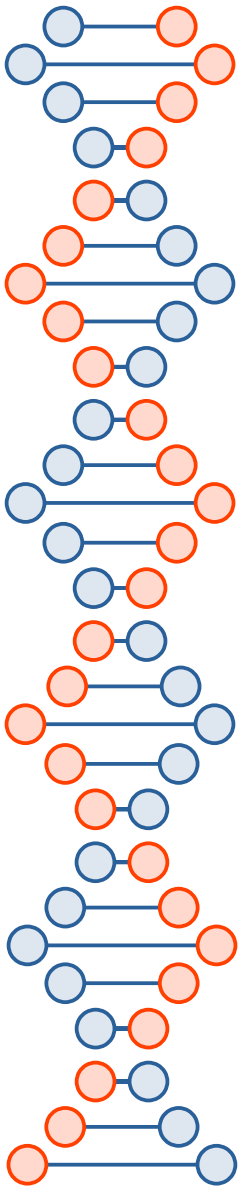
- Shiva modules are loaded from: **“/opt/shiva/modules”**
- Similar to LKM (Loadable kernel module):
  - Shiva modules are ELF relocatable object
  - Shiva modules must have an initialization routine

```
int shakti_main(shiva_ctx_t *ctx)
```



Shiva Module: context struct

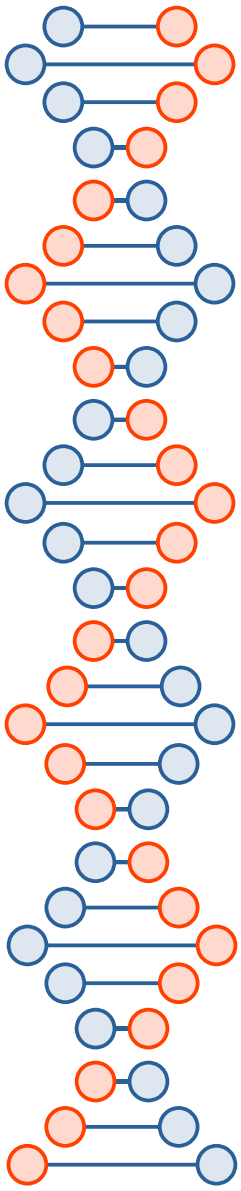
- The context struct "**`shiva_ctx_t`**"
- Passed into the modules init function
- Contains:
  - Register state
  - Process-state
  - Thread-state
  - Control flow data
  - Signal data
  - Shiva-Trace API specific data structures
  - Full access to **`elfobj_t`** of target executable



Shiva modules: Linking

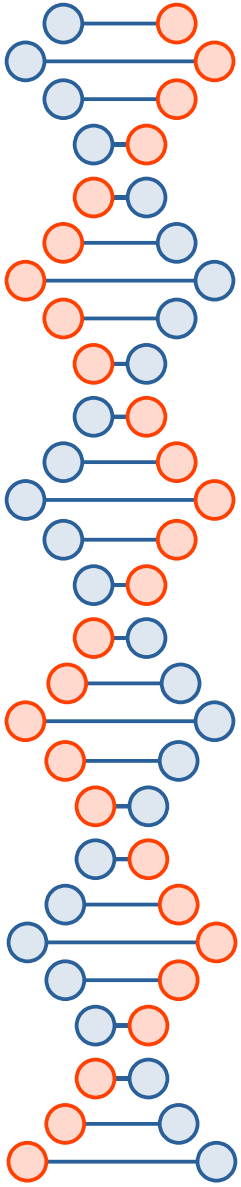
- Shiva modules use the Shiva_trace API for program instrumentation
- Modules are written in C and have access to:
  - **musl-libc**
  - **libelfmaster**
  - **Libudis86**
  - **Shiva-Trace API**

- Shiva loader handles loading and linking of modules
- The module is an **ELF ET_REL** (relocatable object file)
- Sections are mapped appropriately into memory segments: Text, and Data



Shiva modules execute in-process

- Shiva modules are executed **in-process.**
- **In-process** means:
  - The Shiva runtime engine is in the same process as the target program
  - The module also executes within the same process address space
- Instrumentation and hooking is extremely fast
- Module has an ***initialization function***
- Module init function sets hooks and breakpoints within the target program
- Handler functions are callbacks for hooks and breakpoints



Understanding Shiva modules: **gcc code models**

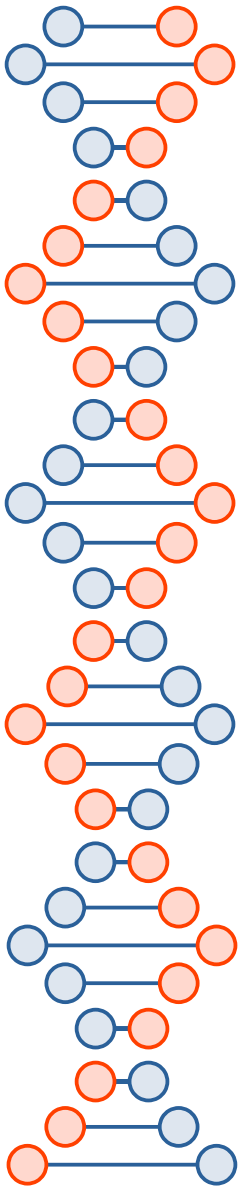
- Shiva modules are currently compiled with GCC and must be compiled with a large code model (`i.e.`

  `gcc -c -mcmodel=large`)

- A large code model is necessary to handle module relocation offsets that exceed 4GB
- On 64bit system: The Shiva executable, the Shiva module, and the target executable will be mapped to addresses that are vastly far apart
- In some instances we can and must use a small code model for modules (More on this later)

A security module. Prevent command injection via system function

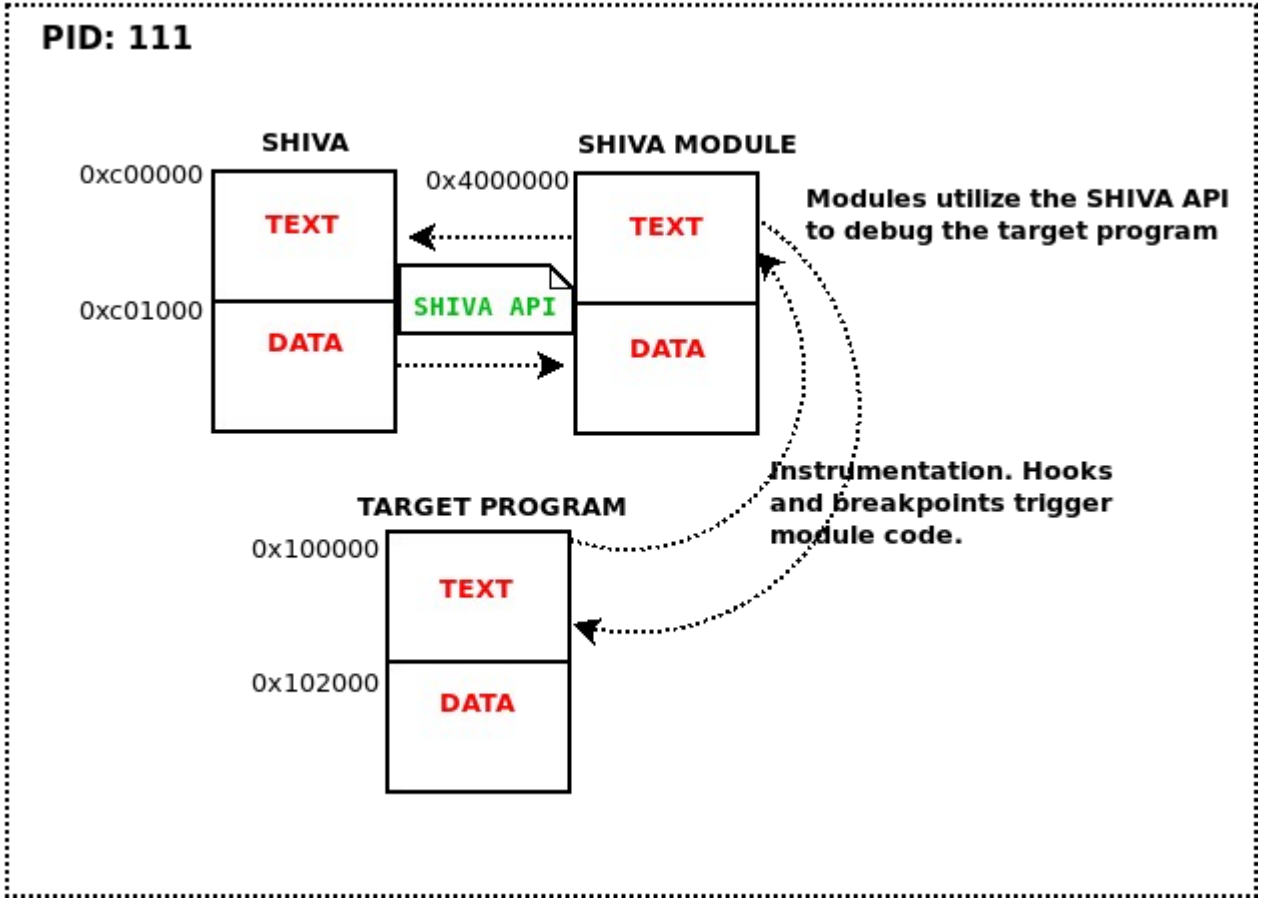
```
#include "../shiva.h"

int n_system(const char *s)
{

        printf("s: %s\n", s);
        if (strstr(s, ";") != NULL || strstr(s, "|") != NULL) {
                printf("Detected possible OS command injection attack '%s'\n", s);
                abort();
        }
        return system(s);
}

int
shakti_main(shiva_ctx_t *ctx)
{
        bool res;
        shiva_error_t error;

        res = shiva_trace(ctx, 0, SHIVA_TRACE_OP_ATTACH,
            NULL, NULL, 0, &error);
        if (res == false) {
                printf("shiva_trace failed: %s\n", shiva_error_msg(&error));
                return -1;
        }
        res = shiva_trace_register_handler(ctx, (void *)&n_system,
            SHIVA_TRACE_BP_PLTGOT, &error);
        if (res == false) {
                printf("shiva_register_handler failed: %s\n",
                    shiva_error_msg(&error));
                return -1;
        }
        res = shiva_trace_set_breakpoint(ctx, (void *)&n_system,
            0, "system", &error);
        if (res == false) {
                printf("shiva_trace_set_breakpoint failed: %s\n", shiva_error_msg(&error));
                return -1;
        }
        return 0;
}
```



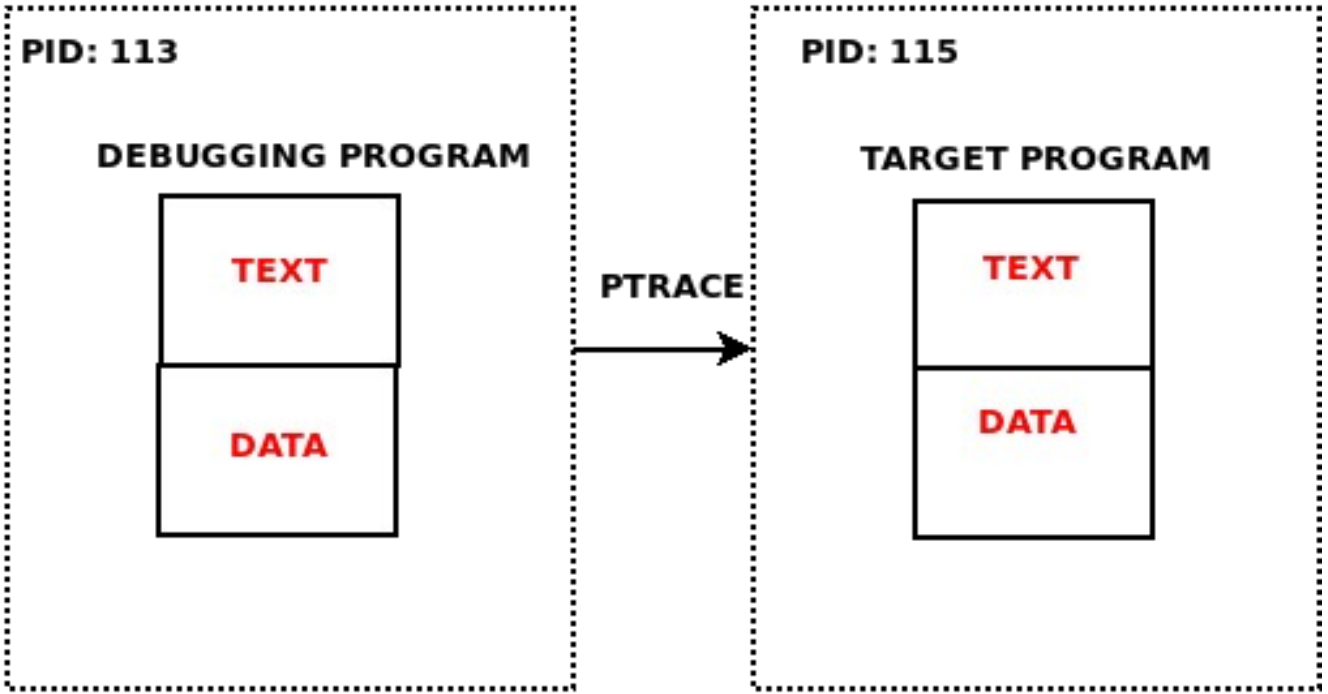
The Shiva API (shiva_trace)

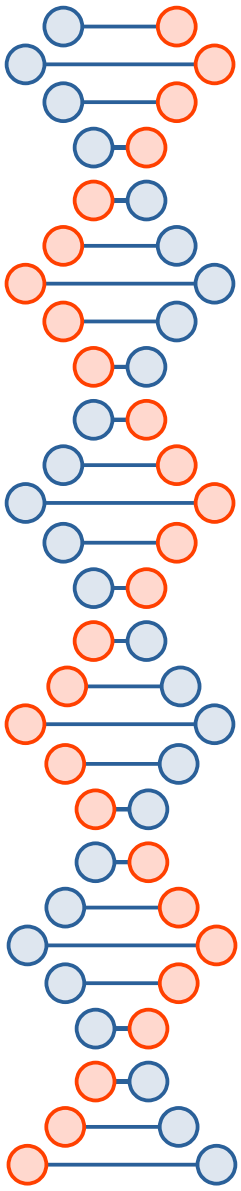
- Shiva modules are written in C
- The Shiva_trace API can: **access, read, transform and instrument the process image.**
- Rich API for process transformation
- Excellent for writing debuggers, tracers, security modules, etc.
- In very early development phase

Shiva debugging model



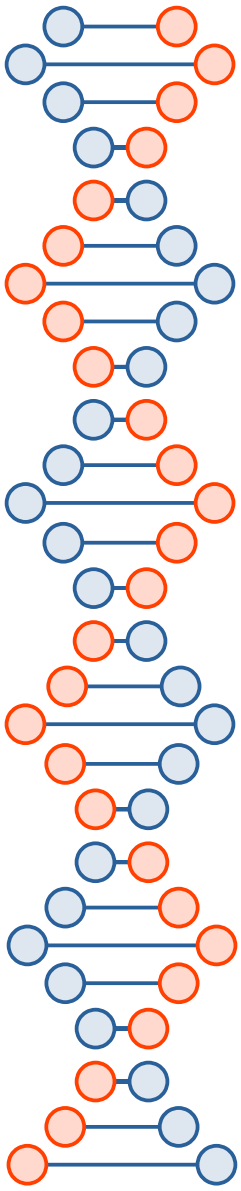
PTRACE Debugging model





Shiva-Trace vs. PTRACE

- The Shiva-Trace API does not rely on the **PTRACE** system call
- Shiva-Trace is effective against hardened binaries and anti-debugging
- Shiva-Trace is implemented as an “in-process” programmable runtime engine.
- **PTRACE** on the other hand, executes from a separate process context.
- **PTRACE**: Slow. And is useless against many Linux anti-debugging techniques.
- **PTRACE** operations send a **SIGSTOP** to the target process.
- **PTRACE** is a system call and requires a context switch, whereas Shiva_trace is implemented purely in userland and less expensive.

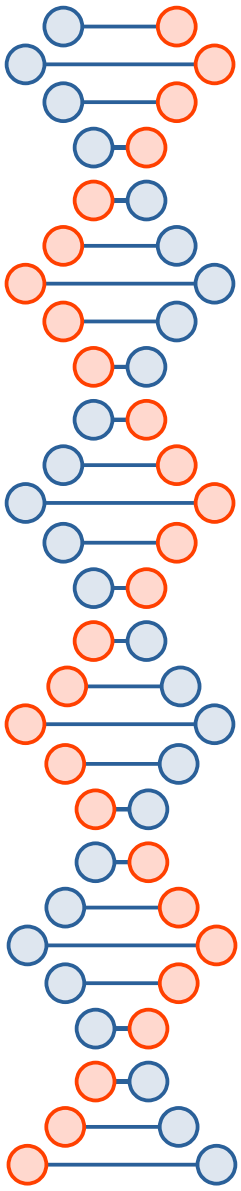


Shiva Trace p3.

- shiva_trace can cleverly instrument the target program at runtime
- shiva_trace uses the terminology **“Hooks”** and **“Breakpoints”** interchangeably.
- Many breakpoints are really just various types of hooks
- shiva_trace can defeat anti-debugging techniques
- Powerful for debugging hostile binaries

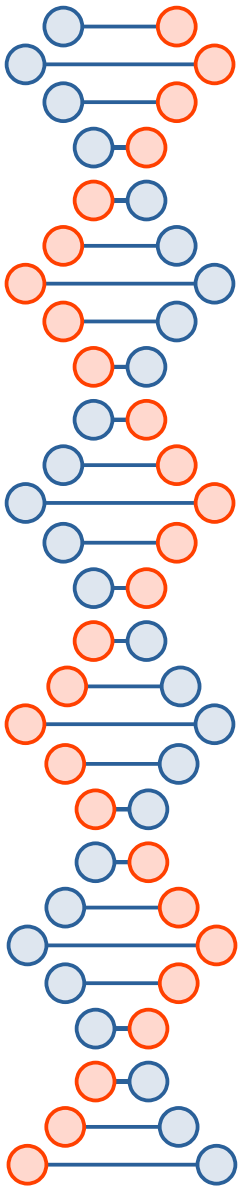
Shiva API basics: shiva_trace()

```
/*
 * shiva_trace lets the module attach to the process, read and write to the process,
 * get and set the register state.
 */
bool shiva_trace(shiva_ctx_t *, pid_t, shiva_trace_op_t, void *, void *, size_t, shiva_error_t *);


typedef enum shiva_trace_op {
        SHIVA_TRACE_OP_CONT = 0,
        SHIVA_TRACE_OP_ATTACH,
        SHIVA_TRACE_OP_POKE,
        SHIVA_TRACE_OP_PEEK,
        SHIVA_TRACE_OP_GETREGS,
        SHIVA_TRACE_OP_SETREGS,
        SHIVA_TRACE_OP_SETFPREGS,
        SHIVA_TRACE_OP_GETSIGINFO,
        SHIVA_TRACE_OP_SETSIGINFO
} shiva_trace_op_t;


~
~
~
~
~
```



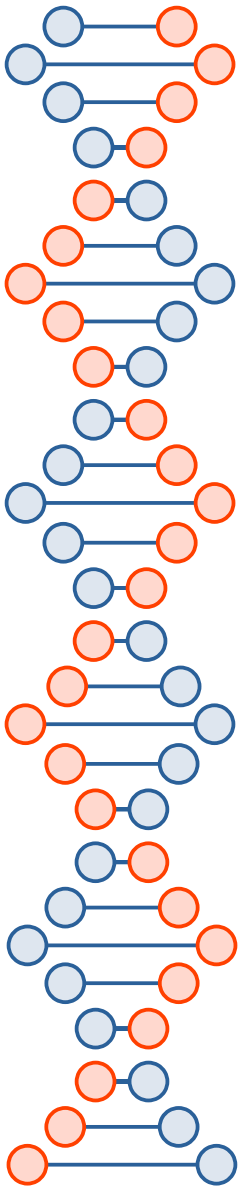
Shiva-Trace API - Breakpoints

- Breakpoints are registered to a handler function
- Similar to Linux Kprobes, but for userland
- Multiple breakpoints can be set for a single handler
- Some breakpoints are hooks
- Some breakpoints are signal driven
- A complex series of in-process trampolines



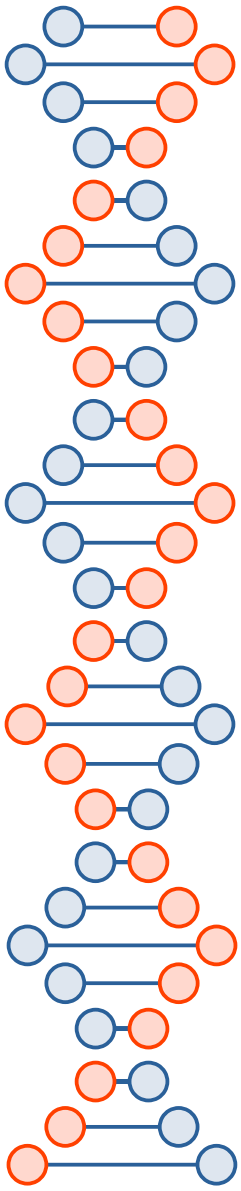
Shiva breakpoints and signals

- We do not use PTRACE to catch signals
- Our debugging module exists within the same PID as the program it's debugging
- Signal driven breakpoints (i.e. **`int3`**) register a handler internally with **`sigaction()`**
- Shiva internally hooks **`signal()`** and **`sigaction()`** in the target program to prevent the target from overwriting breakpoint handlers



A look at all Shiva breakpoint types

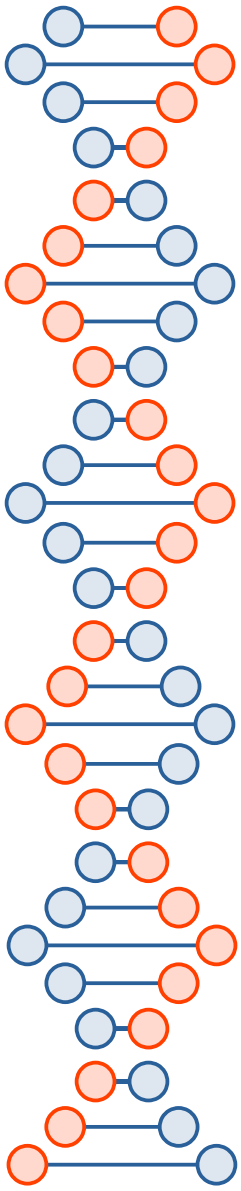
```
typedef enum shiva_trace_bp_type {
        SHIVA_TRACE_BP_JMP = 0,
        SHIVA_TRACE_BP_CALL,
        SHIVA_TRACE_BP_INT3,
        SHIVA_TRACE_BP_SEGV,
        SHIVA_TRACE_BP_SIGILL,
        SHIVA_TRACE_BP_TRAMPOLINE,
        SHIVA_TRACE_BP_PLTGOT
} shiva_trace_bp_type_t;
```



Setting hooks and breakpoints

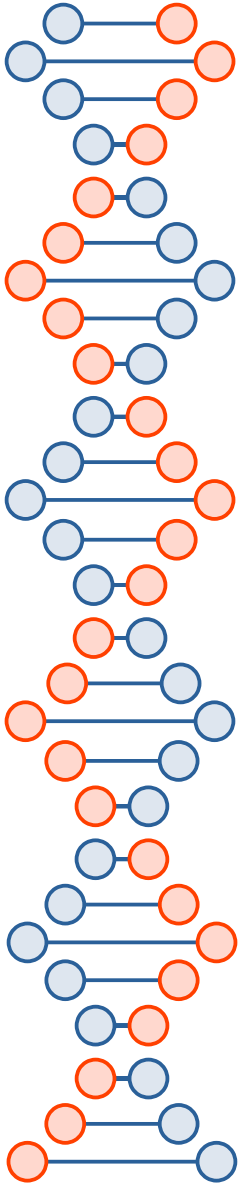
- Breakpoints/hooks are registered to a handler function

```
res = shiva_trace_register_handler(ctx, (void *)&bp_handler,
    SHIVA_TRACE_BP_INT3, &error);
if (res == false) {
        printf("shiva_register_handler failed: %s\n",
            shiva_error_msg(&error));
        return -1;
}
if (elf_symbol_by_name(&ctx->elfobj, "main", &symbol) == false) {
        printf("couldn't find sybol main in target\n");
        return -1;
}
res = shiva_trace_set_breakpoint(ctx, (void *)&bp_handler,
    symbol.value + ctx->ulexec.base_vaddr, NULL, &error);
if (res == false) {
        printf("shiva_trace_set_breakpoint failed: %s\n", shiva_error_msg(&error));
        return -1;
}
```



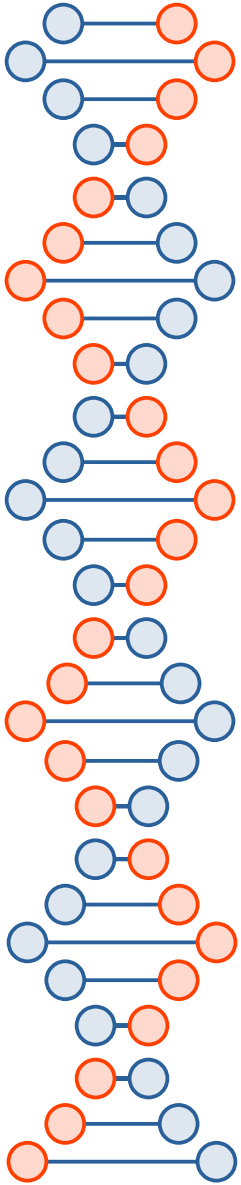
Basic handler function

```
void
bp_handler(int sig, siginfo_t *si, void *data)
{

        ucontext_t *uctx = (ucontext_t *)data;

        printf("Breakpoint at %#lx\n", uctx->uc_mcontext.gregs[REG_RIP] - 1);
        return;
}
```



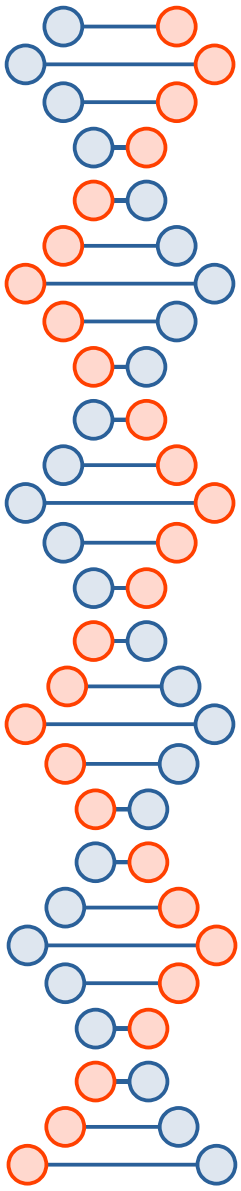
SHIVA_TRACE_BP_JMP

- **Breakpoint type**: Control flow hook
- Set on 5 byte jmp instructions
- The jmp offset is replaced by an offset to your registered handler function
- When done, the handler jumps back to the original target.
- **`SHIVA_TRACE_LONGJMP_RETURN`** macro: rewinds the stack and jmp's to the original offset
- Defeats breakpoint detection with anti-debugging



SHIVA_TRACE_BP_CALL

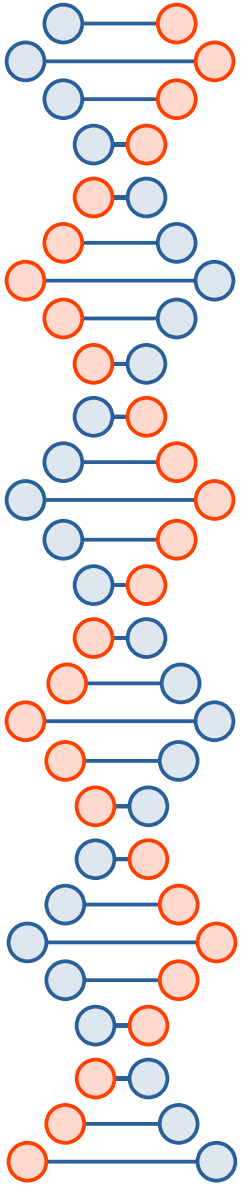
- **Breakpoint type**: Control flow hook
- Set on 5 byte immediate call instructions
- Original call target is replaced with offset to registered handler
- The registered handler returns after calling the original function (If desired)
- Extremely fast way to build a function tracer: (See modules/examples/func_tracer.c)
- Only works when executing Shiva directly (Not as an interpreter) due to required small-code model.
- Defeats breakpoint detection



SHIVA_TRACE_BP_TRAMPOLINE

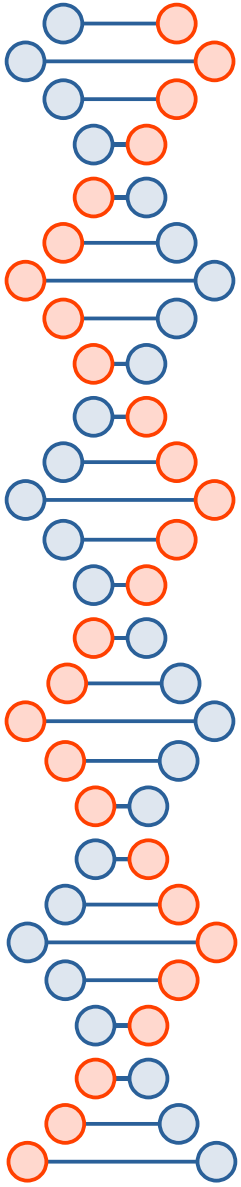
- **Breakpoint type:** Control flow hook
- This type of breakpoint inserts a trampoline on the entry point of the function
  - **movq $target_addr, %rax**

    jmp *%rax
- Excellent for hooking any function locally or globally within the address space
- Trampoline function restores the original code bytes before calling the original function
- Not currently thread-safe. Memcpy's that place and restore hooks are ***not atomic operations***
- Works well in a large code model whereas ***call hooks*** do not.



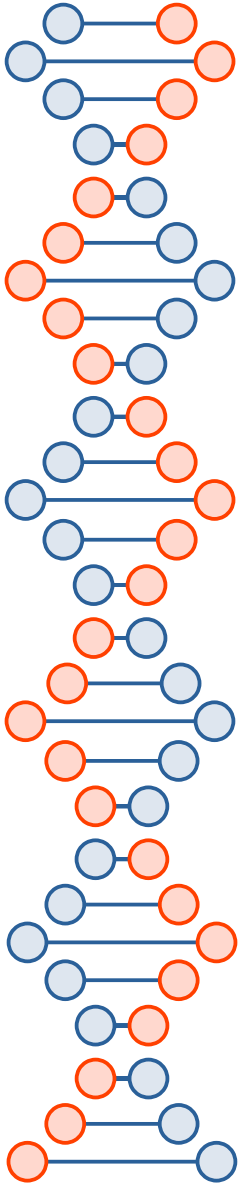
SHIVA_TRACE_BP_PLTGOT

- Hook any shared library function within the PLT/GOT
- Modifies the **`.got.plt[entry]`** of the given symbol
- Was tricky to implement. The RTLD wants to patch over our GOT patch. (i.e. with **`DT_BINDNOW`**)
- Very low overhead, and thread safe
- Handler is usually a hook-replacement function for the function being hooked
- A single handler can also be used for multiple PLTGOT hooks. (*i.e. Any PLT call triggers the same handler*)



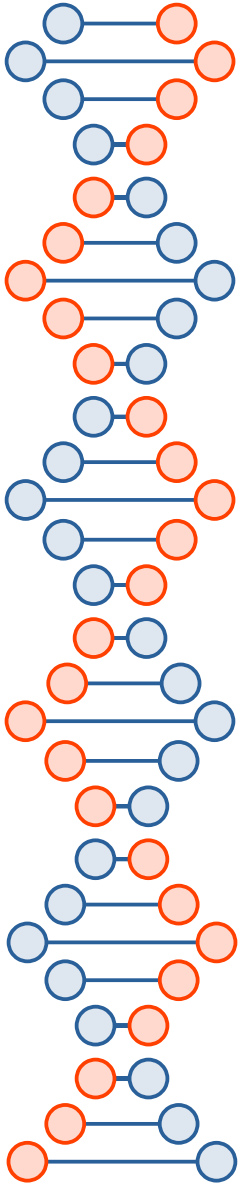
SHIVA_TRACE_BP_INT3

- **Breakpoint type:** Signal driven
- Traditional **`int3`** breakpoint uses **`0xcc`** opcode
- Causes kernel to deliver **`SIGTRAP`**
- The breakpoint handler is essentially a signal handler
- Uses **`struct ucontext`** to get register state in this case, as any signal handler would
- If the target program already had a **`SIGTRAP`** handler: ours replaces it, and optionally can invoke the original handler



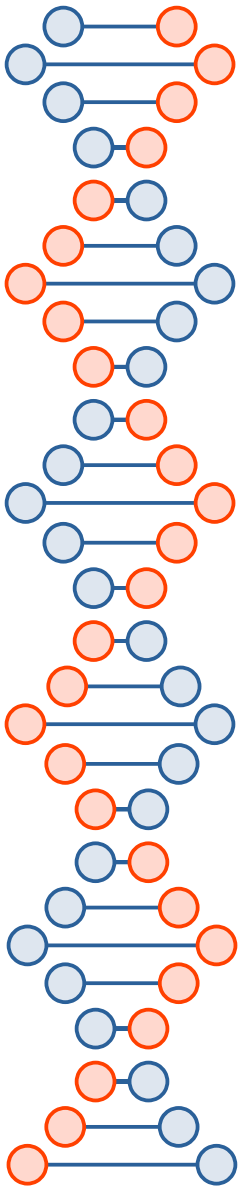
SHIVA_TRACE_BP_SIGILL

- **Breakpoint type:** Signal driven
- This breakpoint inserts a two byte **`ud2`** (undefined instruction) to trigger **`SIGILL`**. **`"\x0f\x0b\"`**
- Breakpoint is registered to your signal handler function
- Can be helpful against hardened binaries that perform anti-debugging on standard breakpoints



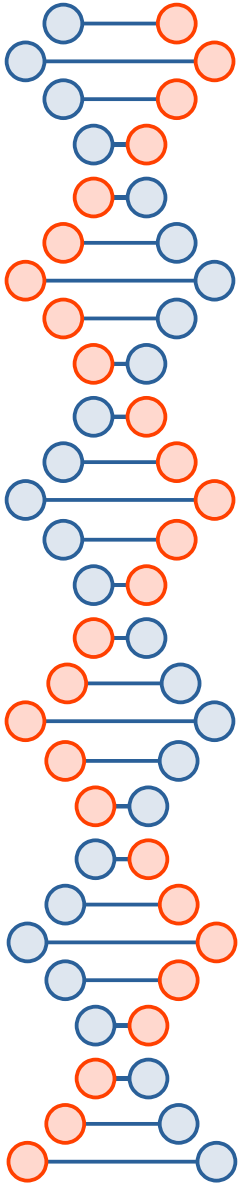
SHIVA_TRACE_BP_SEGV

- **Breakpoint type:** Signal driven
- Breakpoint code:
  - **`push <invalid_offset>(%rip)`**
- Shiva_trace API inserts an instruction with an invalid offset or address
- Causes an invalid memory dereference, and thus a **`SIGSEGV`**
- The handler function is a signal handler that catches the **`SEGV,`** and executes your custom code



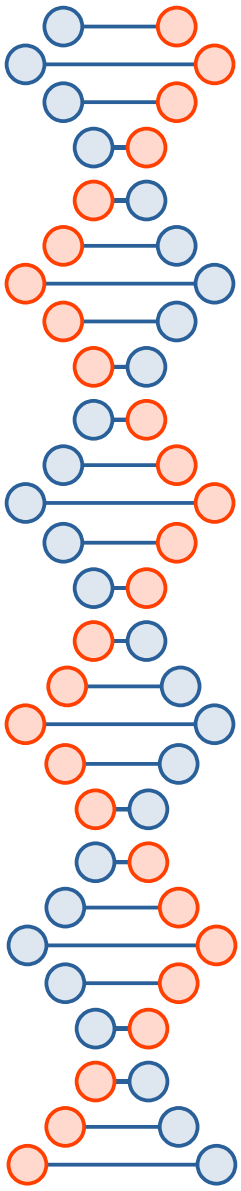
Reading and writing to process memory

- The shiva_trace() API call
- **`SHIVA_TRACE_OP_PEEK`**
  - Allows the module to read anywhere in process memory
- **`SHIVA_TRACE_OP_POKE`**
  - Allows the module to write anywhere in process memory (Except to the Shiva code itself)



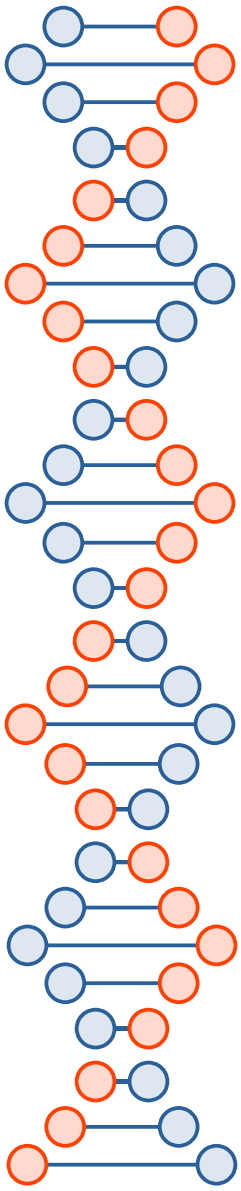
Getting and setting register state

- The shiva_trace API call
- **`SHIVA_TRACE_OP_GETREGS`**
  - Get the current register values
- **`SHIVA_TRACE_OP_SETREGS`**
  - Set the register values



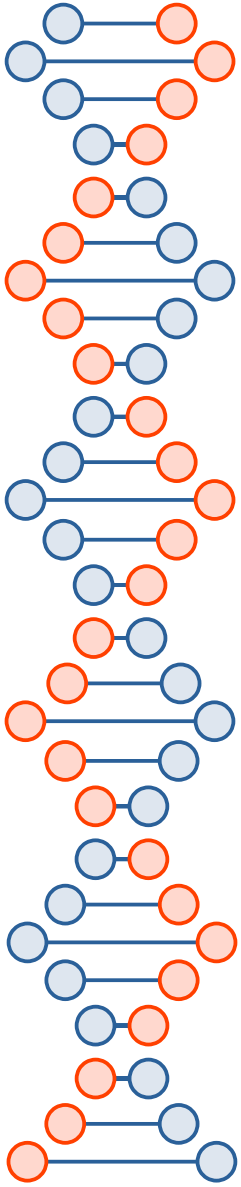
In-process debugging tools with Shiva

- When running Shiva directly it can be used for creating powerful debugging tools
- “In-process” debugging technology is extremely fast
- Less context switching, more in-process trampolines
- **`modules/examples/func_tracer.c`** is a module for function tracing
- Sets a **`SHIVA_TRACE_BP_CALL`** hook on every immediate call
- Registered handler function simply prints each function name being called



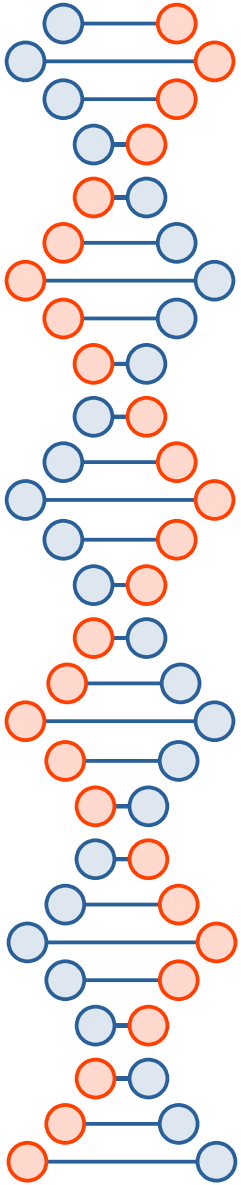
Shiva module: func_tracer.o

- func_tracer.c
- Only 84 lines of code
- Much faster than tracers that are written using **`PTRACE`**
- Benchmarks demonstrating performance:
- Command **`./test`**:
  - **`Real 0m0.052s`**
- Tracing **`./test`** with Shiva:
  - **`Real 0m0.155s`**
- Tracing **`./test`** with **`/bin/ltrace`**
  - **`Real 0m3.582`**



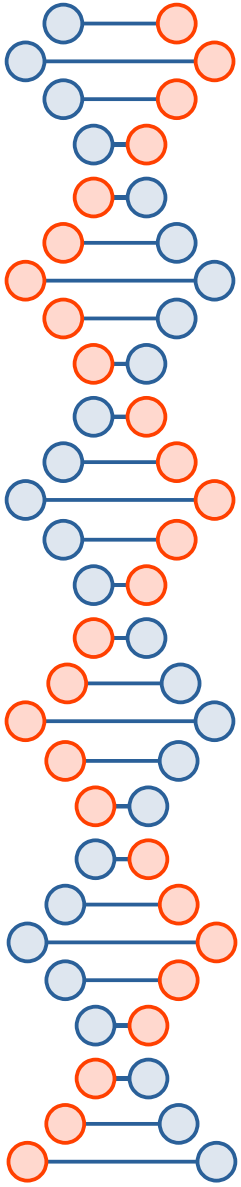
Control flow integrity: ret2PLT protection

- **`modules/examples/plt_cfi.c`**
- Only 109 lines of code
- Forward edge / Backward edge CFI
- Sets a **`SHIVA_TRACE_BP_PLTGOT`** hook on every PLT call
- **`shiva_trace_set_breakpoint()`** installs hooks/breakpoints and stores CFI data in breakpoint struct
- Every PLT call is redirected to our modules handler:
  `int plt_handler(void *arg)`



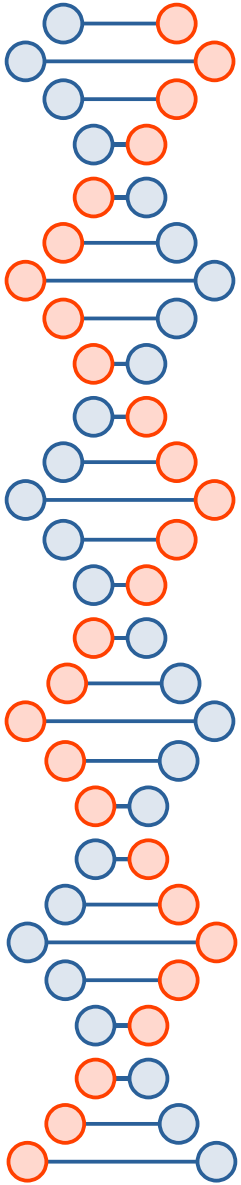
Ret2plt handler function

- **`int plt_handler(void *arg)`**
- Checks the return address
- Prevents arbitrary calls to the PLT
- A ret2PLT attack will fail against this



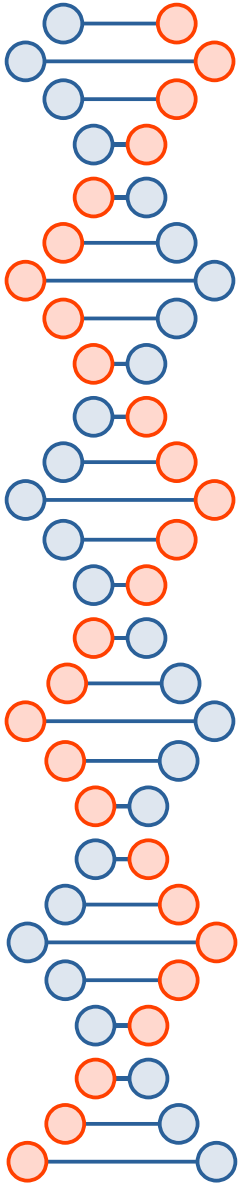
Mitigate OS command injection

- **`modules/examples/os_inject.o`**
- Only 48 lines of code
- **`SHIVA_TRACE_BP_PLTGOT`** hook on **`system()`** function
- Replaced with secure function **`n_system()`**
- **`n_system()`** sanitizes the command string



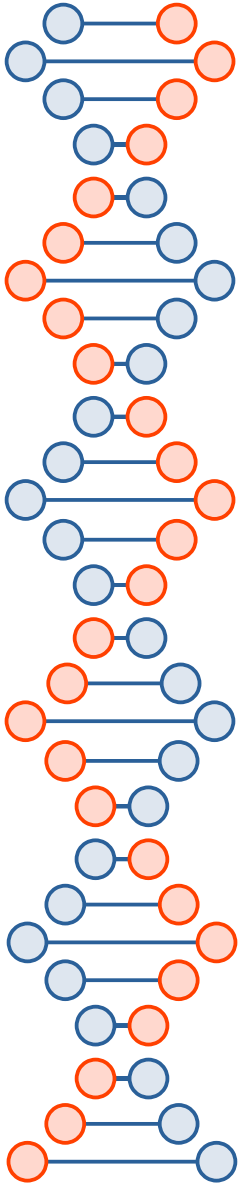
Crack software

- ./crackme
- Shiva module to crack serial number validator
- **`modules/examples/crackme_bypass.o`**
- Redirects: **`int check_serial(char *serial)`**
- To a modified version of the function
- Uses a **`SHIVA_TRACE_BP_TRAMPOLINE`** breakpoint



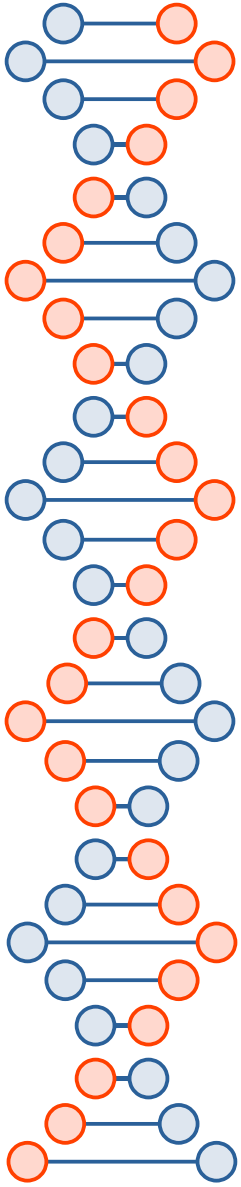
Shiva for debugging hardened binaries

- There is a long history of anti-debugging techniques
- PTRACE system call is easily foiled
- PTRACE fails on PaX hardened binaries
  - **`mprotect`** restrictions prevent **`ptrace(PTRACE_POKETEXT, …);`**
- Shiva maps the hostile/hardened program into it's own address space
- In-process debugging can be very effective for Reverse engineering tough problems



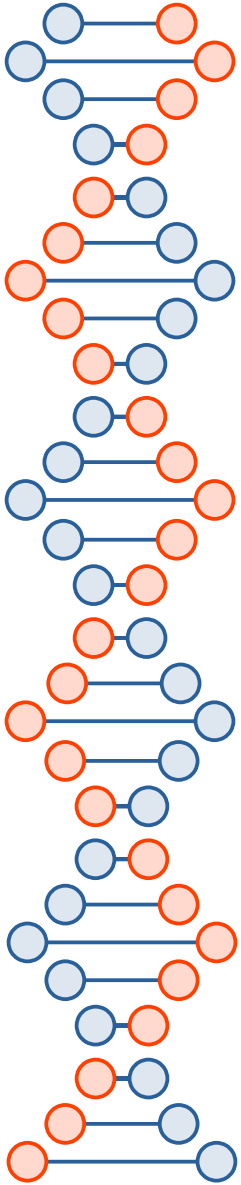
Tracing a hardened binary

- **`modules/examples/func_tracer.o`**
- Capable of tracing binaries with anti-debugging
- **`test_antidebug.c`**: Simple anti-debugging techniques
  - **`ptrace(PTRACE_TRACEME, 0, 0, 0, 0);`**
  - **`prctl(PR_SET_DUMPABLE, 0, 0, 0, 0);`**



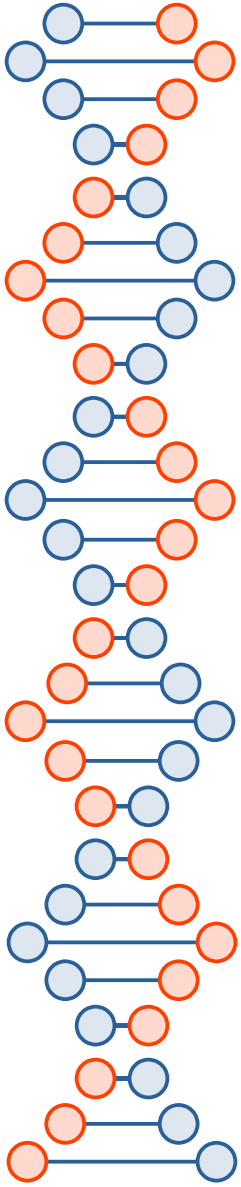
Shiva works on stripped ELF binaries

- Corrupted meta-data
- Missing symbol tables
- Missing section header table
- Shiva uses libelfmaster: https://github.com/elfmaster/libelfmaster
- Forensic reconstruction of stripped binaries:
  - Symbol table reconstruction
  - Section header table reconstruction



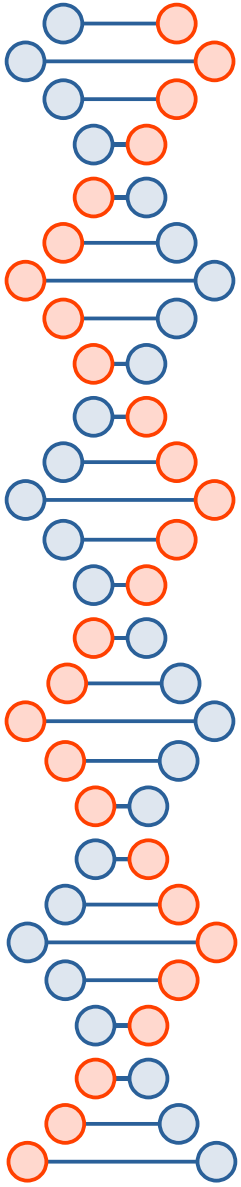
Tracing a stripped binary

- ltrace fails to get any tracing info:
  - Requires section headers and symbol tables
- Our Shiva module for function tracing succeeds
  - Forensically reconstructs section and symbol data



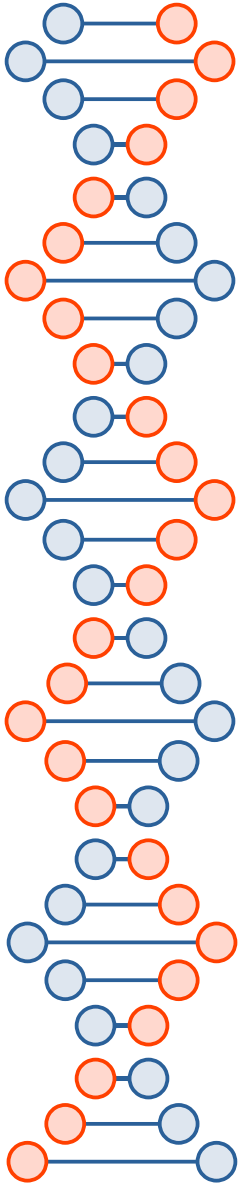
Current limitations

- Support for multi-threaded applications
- Shiva doesn't work on ELF's with **`-fcf-protection`**
- Shiva can load only one module at a time
- Shiva only works with **`x86_64 ET_DYN`** binaries
  - Will add support for **`x86_64 ET_EXEC`**
- Running Shiva as an interpreter requires a large code model for all modules
  - Some Shiva features require a small code model (i.e. **`SHIVA_TRACE_BP_CALL`** breakpoints



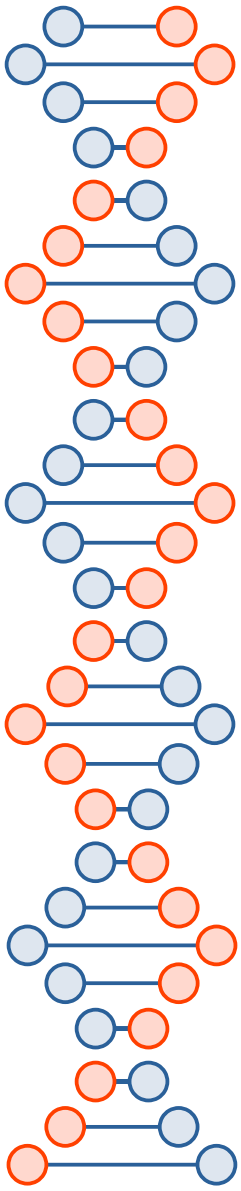
Future plans

- Open source the technology
- Continue to develop and expand the usability and functionality
- **shiva-gcc** wrapper for modules
- Design a suite of security modules for process hardening
- Handle loading multiple modules
- Handle threads



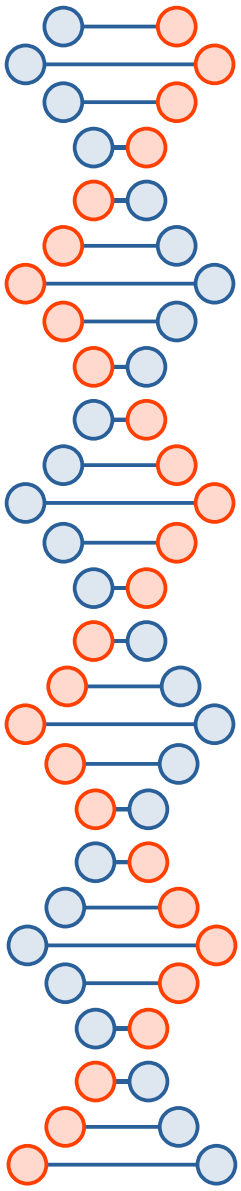
A note on handling threads

- How do we handle threads without using SYS_ptrace?
- TLS resolver hooks
- pthread_create hooks
- SYS_clone hooks
- Shared memory segments



Related works

- Linux kprobe instrumentation
  - http://phrack.org/issues/67/6.html
- Preloading the linker for fun and profit
  - https://tmpout.sh/2/6.html
- Embedded ELF debugging (by Mayhem & ERESI)
  - http://phrack.org/issues/63/9.html
- Userland exec implementation (By the Grugq:)
  - https://grugq.github.io/docs/ul_exec.txt



Questions?

- Ryan@bitlackeys.org
- Bitlackeys.org
- https://github.com/elfmaster/shiva