patriksimek / **vm2** Public

Code Issues 10 Pull requests 1 Actions Projects Wiki Secur

Commit `f9b700b`

patriksimek committed last week

```
fix(GHSA-grj5-jjm8-h35p): block Array species self-return sandbox escape

V8's ArraySpeciesCreate reads `this.constructor[Symbol.species]` directly
on raw objects, bypassing the bridge proxy. Two-layer defense in
lib/bridge.js: (1) the proxy `get` trap for `constructor` on host arrays
returns a cached Array reference captured at module load
(prototype-pollution-proof); (2) the `apply`/`construct` traps neutralise
every host array reachable as context or argument by shadowing
`constructor = undefined` before the host call and restoring in `finally`.
Non-configurable or non-extensible arrays with attacker state are rejected
with VMError. See test/ghsa/GHSA-grj5-jjm8-h35p/ for repros.
```

main · v3.11.2 ••• v3.11.0

1 parent 8dd0591 commit f9b700b

3 files changed +498 -63

Top

Filter files…

- docs
  - ATTACKS.md
- lib
  - bridge.js
- test/ghsa/GHSA-grj5-jjm8-h35p
  - repro.js

Search within code

docs/ATTACKS.md

```
@@ -1283,11 +1283,14 @@ The `Object.assign` bypass is particularly insidious: the sandbox proxy's `set`

1283 1283
1284 1284  ### Mitigation
1285 1285
1286      - Three-layer defense in `bridge.js`:
     1286 + Two-layer defense in `lib/bridge.js`:
1287 1287
1288      - 1. **Proxy `set` trap**: Intercepts direct `constructor` writes and stores locally on proxy target.
1289      - 2. **Proxy `defineProperty` trap**: Same interception for `Object.defineProperty` from sandbox code.
1290      - 3. **Apply trap species neutralization**: Before and after every host function call, sets `constructor = undefined` as own property on host arrays (context and args). This shadows both own and prototype-inherited constructors. `ArraySpeciesCreate` treats `constructor = undefined` as "use default Array constructor". Non-configurable constructors detected by `Reflect.deleteProperty` returning false, blocked via VMError. Non-extensible arrays detected by `Reflect.defineProperty` returning false, blocked via VMError. `Array.isArray` works cross-realm to identify host arrays.
     1288 + 1. **Proxy `get` trap — cached `Array` ctor for host arrays**: When sandbox code reads `.constructor` on a host-array-backed proxy, the trap returns a module-load-time-captured `thisArrayCtor = Array` reference. This bypass of the normal property read neutralises any attacker-installed `constructor` (direct `r.constructor = x`, `Object.defineProperty`, `Object.assign`, prototype-chain injection via `Object.setPrototypeOf`) and is immune to prototype pollution of `Array.prototype.constructor`. Only defends sandbox-side reads; does not cover V8-internal reads issued from the host realm.
     1289 + 2. **Apply/construct trap neutralize-and-restore**: Before every `otherReflectApply(object, context, args)` and `otherReflectConstruct(object, args)` — i.e. every sandbox→host function invocation — the bridge walks `context` and each top-level argument. For every host array found (`Array.isArray` is cross-realm safe), it installs `constructor = undefined` as a data own property (shadowing both own and inherited constructors; the ES2024 spec explicitly maps `constructor === undefined` to `%Array%` in ArraySpeciesCreate). After the host call returns — in a `finally` — the prior descriptor is restored (or the shadow deleted if none existed). This covers V8-internal reads issued from the host realm during the call.
     1290 +
```

```
                1291 + Both layers reject un-neutralisable arrays with `VMError`: a pre-installed
                       non-configurable `constructor` whose value is anything other than
                       `undefined`, or a non-extensible array without an own `constructor` slot,
                       cannot be safely shadowed or restored and is treated as an attack.
                1292 +
                1293 + The neutralize-on-entry/restore-on-exit pattern mirrors `resetPromiseSpecies`
                       in `setup-sandbox.js`, which closes the equivalent V8-internal-bypass class
                       for Promise.
1291    1294
1292    1295    ### Detection Rules
1293    1296
```

lib/bridge.js

```
                @@ -145,6 +145,13 @@ const thisMapSet = ThisMap.prototype.set;
145     145     const thisFunction = Function;
146     146     const thisFunctionBind = thisFunction.prototype.bind;
147     147     const thisArrayIsArray = Array.isArray;
                148 + // SECURITY (GHSA-grj5-jjm8-h35p): Cache the this-realm Array constructor at
                       module
                149 + // load time, BEFORE any sandbox code runs. Used as the safe species-
                       neutralized
                150 + // return value from proxy.get 'constructor' on host arrays. We cache this
                       directly
                151 + // from `global.Array` so that prototype pollution attacks
                152 + // (e.g., `Array.prototype.constructor = attackerFn` via cross-realm proto
                       injection)
                153 + // cannot redirect our defense to an attacker-controlled value.
                154 + const thisArrayCtor = Array;
148     155     const thisErrorCaptureStackTrace = Error.captureStackTrace;
149     156
150     157     const thisSymbolToString = Symbol.prototype.toString;
                @@ -364,58 +371,12 @@ function createBridge(otherInit, registerProxy) {
364     371             if (!thisReflectPreventExtensions(target)) throw thisUnexpected();
365     372         }
366     373
367             -     /**
368             -      * Neutralize Array species on a host-realm value.
369             -      *
```

```
370 -       * V8's ArraySpeciesCreate algorithm reads `obj.constructor[Symbol.species]` on
371 -       * the raw host object, bypassing proxy traps. If an attacker sets constructor
372 -       * to a function that returns the same array (species self-return), map/filter/etc.
373 -       * store raw host values directly into that array, bypassing bridge sanitization.
374 -       *
375 -       * This function sets `constructor = undefined` as an own data property on any
376 -       * host array. With `constructor` undefined, ArraySpeciesCreate falls back to
377 -       * the default Array constructor, which is safe.
378 -       *
379 -       * Called before and after every host function call in the apply trap.
380 -       */
381 -      function neutralizeArraySpecies(value) {
382 -          // Only process non-null objects (arrays are objects)
383 -          if (value === null || typeof value !== 'object') return;
384 -          try {
385 -              // Array.isArray works cross-realm — identifies host arrays correctly
386 -              if (!thisArrayIsArray(value)) return;
387 -
388 -              // Set constructor = undefined as own data property.
389 -              // This shadows any inherited or attacker-set constructor.
390 -              const success = otherReflectDefineProperty(value, 'constructor', {
391 -                  __proto__: null,
392 -                  value: undefined,
393 -                  writable: true,
394 -                  configurable: true
395 -              });
396 -              if (!success) {
397 -                  // If defineProperty failed, the array may be non-extensible or
398 -                  // constructor is non-configurable (attacker froze it).
399 -                  // Either way, throw to prevent the call from proceeding.
400 -                  throw new VMError('Cannot neutralize array species: constructor is non-configurable or array is non-extensible');
401 -              }
```

```
402         -         } catch (e) {
403         -             if (e instanceof VMError) throw e;
404         -             // Swallow other errors (e.g., from Proxy traps on exotic objects)
405         -         }
406         -     }
407         -
408         -     /**
409         -      * Neutralize Array species on all arguments and context before/after a host call.
410         -      */
411         -     function neutralizeArraySpeciesArgs(context, args) {
412         -         neutralizeArraySpecies(context);
413         -         if (args) {
414         -             for (let i = 0; i < args.length; i++) {
415         -                 neutralizeArraySpecies(args[i]);
416         -             }
417         -         }
418         -     }
        374 +     // (Removed PR #563's `neutralizeArraySpecies` / `neutralizeArraySpeciesArgs`
        375 +     // helpers — superseded by `neutralizeArraySpeciesOn` / `neutralizeArraySpeciesBatch`
        376 +     // + `restoreArraySpeciesOn` / `restoreArraySpeciesBatch` defined below, which
        377 +     // add restore-on-exit so host arrays' `constructor` isn't permanently mutated.
        378 +     // The PR's set/defineProperty trap interception of `constructor` writes is
        379 +     // preserved as a complementary defense layer.)
419     380
420     381         function thisAddProtoMapping(proto, other, name) {
421     382             // Note: proto@this(unsafe) other@other(unsafe) name@this(unsafe) throws@this(unsafe)
            @@ -602,6 +563,187 @@ function createBridge(otherInit, registerProxy) {
602     563             }
603     564         }
604     565
        566 +     // SECURITY (GHSA-grj5-jjm8-h35p): Array species self-return escape defense.
        567 +     //
```

```
568 +     // INVARIANT: When the bridge invokes a host function, no host-realm array used as
569 +     // `this` (context) or as an argument may have an attacker-controlled `constructor`
570 +     // property visible to V8's internal ArraySpeciesCreate during the call.
571 +     //
572 +     // WHY: V8's ArraySpeciesCreate (used by Array#map/filter/slice/concat/splice and
573 +     // TypedArray equivalents) reads `this.constructor[Symbol.species]` DIRECTLY on the
574 +     // raw host object, completely bypassing our proxy traps. If an attacker can install
575 +     // a sandbox function `x` with `x[Symbol.species] = x` as the host array's constructor,
576 +     // then call `r.map(f)` through the bridge, V8 will call `new x(len)` (returning `r`
577 +     // itself) and store mapped values directly into `r` via CreateDataPropertyOrThrow,
578 +     // bypassing the bridge entirely -- the attacker reads them back from `r`.
579 +     //
580 +     // The prior fixes (ebcfe94, 9084cd6) blocked the Function-constructor exfiltration
581 +     // chain that this primitive was originally composed with, but the primitive itself
582 +     // -- writing raw host values into a sandbox-visible slot -- remained a bridge bypass.
583 +     // This defense closes the class.
584 +     //
585 +     // Strategy:
586 +     //   1. Before every sandbox->host function invocation (apply/construct traps), walk
587 +     //      `context` and each element of `args`. For every host array found, neutralize
588 +     //      its species state by installing `constructor = undefined` as a data own
589 +     //      property. An own-property `undefined` shadows any own or inherited ctor, and
590 +     //      ArraySpeciesCreate treats undefined as "use the default %Array% constructor"
591 +     //      (spec ES2024 23.1.3.1 step 3), producing a fresh plain array.
```

```
592 +     //   2. After the call completes (finally), restore the original state: if the array
593 +     //      had an own `constructor` descriptor before, re-install it; otherwise delete
594 +     //      our shadow.
595 +     //   3. If any host array is non-extensible or has a non-configurable `constructor`
596 +     //      that isn't already `undefined`, reject the call with VMError rather than
597 +     //      proceeding with an un-neutralizable species channel.
598 +     //
599 +     // This neutralize-on-entry/restore-on-exit pattern is analogous to resetPromiseSpecies
600 +     // in setup-sandbox.js, which defends the same V8-internal-bypass class for Promises.
601 +
602 +     // SECURITY: Sentinel used to tag saved-state records. Using a unique module-local
603 +     // object ensures we never confuse attacker-installed state with our own.
604 +     const SPECIES_NEUTRALIZED = {__proto__: null};
605 +
606 +     // SECURITY: Neutralize the ArraySpeciesCreate channel on a single host array.
607 +     // Returns an opaque saved-state record to be passed to restoreArraySpeciesOn,
608 +     // or null if `arr` did not need neutralization (not an array / not host-realm).
609 +     // Throws VMError if the array's species state cannot be safely neutralized
610 +     // (non-configurable attacker-installed constructor, non-extensible with a
611 +     // constructor own property, etc.).
612 +     function neutralizeArraySpeciesOn(arr) {
613 +         // SECURITY: only host-realm raw arrays need neutralization. Sandbox arrays
614 +         // that cross back are already fresh objects produced by thisFromOtherArguments
615 +         // (which creates them with thisReflectDefineProperty in the local realm) --
616 +         // V8 internal reads on those happen in the local realm too, and the
617 +         // local-realm Array.prototype.constructor is not attacker-controlled.
618 +         if (arr === null || typeof arr !== 'object') return null;
```

```
619 +         let isHostArray;
620 +         try {
621 +             // SECURITY: thisArrayIsArray works cross-realm -- Array.isArray
      returns
622 +             // true for any ECMAScript Array exotic object regardless of realm.
623 +             isHostArray = thisArrayIsArray(arr);
624 +         } catch (e) {
625 +             return null;
626 +         }
627 +         if (!isHostArray) return null;
628 +
629 +         // SECURITY: capture original own descriptor (if any) before mutating.
630 +         let originalDesc;
631 +         try {
632 +             originalDesc = otherSafeGetOwnPropertyDescriptor(arr,
      'constructor');
633 +         } catch (e) { // @other(unsafe)
634 +             throw thisFromOtherForThrow(e);
635 +         }
636 +
637 +         // SECURITY: if attacker pre-installed a non-configurable
      `constructor`, we
638 +         // cannot safely remove or shadow it for the duration of the call.
      Reject.
639 +         if (originalDesc && originalDesc.configurable === false) {
640 +             // An existing non-configurable `constructor === undefined` data
      property
641 +             // is the already-neutralized shape we want. Anything else is an
      attack.
642 +             if (!(originalDesc.value === undefined && originalDesc.writable ===
      false)) {
643 +                 throw new VMError('Unsafe array constructor cannot be
      neutralized');
644 +             }
645 +             // Already safely neutralized; no-op.
646 +             return null;
647 +         }
648 +
649 +         // SECURITY: if array is non-extensible and has no own `constructor`
      slot, we
```

```
650 +         // cannot install our shadow. The inherited Array.prototype.constructor value
651 +         // is benign (host %Array%), but an attacker may have shadowed it via the
652 +         // prototype chain (setPrototypeOf to an intermediate proto). Reject.
653 +         let isExt;
654 +         try {
655 +             isExt = otherReflectIsExtensible(arr);
656 +         } catch (e) {
657 +             throw thisFromOtherForThrow(e);
658 +         }
659 +         if (!isExt && !originalDesc) {
660 +             throw new VMError('Unsafe non-extensible array passed across bridge');
661 +         }
662 +
663 +         // SECURITY: install `constructor = undefined` as a data own property. This
664 +         // shadows any inherited or attacker-installed constructor; V8's
665 +         // ArraySpeciesCreate treats undefined as "use %Array%", producing a fresh
666 +         // plain array that is NOT the attacker's target.
667 +         let defined;
668 +         try {
669 +             defined = otherReflectDefineProperty(arr, 'constructor', {
670 +                 __proto__: null,
671 +                 value: undefined,
672 +                 writable: true,
673 +                 enumerable: false,
674 +                 configurable: true
675 +             });
676 +         } catch (e) { // @other(unsafe)
677 +             throw thisFromOtherForThrow(e);
678 +         }
679 +         if (!defined) {
680 +             // SECURITY: defineProperty returned false (e.g., frozen object). Reject.
681 +             throw new VMError('Unsafe array state; cannot neutralize species');
682 +         }
683 +
```

```
684 +         return {
685 +             __proto__: null,
686 +             arr: arr,
687 +             originalDesc: originalDesc,
688 +             marker: SPECIES_NEUTRALIZED
689 +         };
690 +     }
691 +
692 +     // SECURITY: Restore the original `constructor` state on a host array after the
693 +     // guarded host call completes. Called from a `finally` block so that errors in
694 +     // the host call do not leave the array in a neutralized state.
695 +     function restoreArraySpeciesOn(saved) {
696 +         if (!saved || saved.marker !== SPECIES_NEUTRALIZED) return;
697 +         const {arr, originalDesc} = saved;
698 +         try {
699 +             if (originalDesc) {
700 +                 // SECURITY: put back exactly what was there before (including any
701 +                 // non-writable/non-enumerable flags). originalDesc has __proto__ null
702 +                 // already (via otherSafeGetOwnPropertyDescriptor).
703 +                 otherReflectDefineProperty(arr, 'constructor', originalDesc);
704 +             } else {
705 +                 // SECURITY: no prior own property -- remove our shadow so inherited
706 +                 // constructor becomes visible again (preserves legitimate API semantics).
707 +                 otherReflectDeleteProperty(arr, 'constructor');
708 +             }
709 +         } catch (e) {
710 +             // SECURITY: swallow restore errors -- the array is in an inert (undefined
711 +             // constructor) state, which is strictly safer than leaving it unrestored
712 +             // if there were an exception. We intentionally do NOT re-throw because
713 +             // this runs in a finally that must not mask the primary error.
714 +         }
```

```
      715  +     }
      716  +
      717  +     // SECURITY: Walk `context` and every element of `args`, neutralize species on
      718  +     // each host array found, return a flat list of saved-state records. Used by the
      719  +     // apply/construct traps. The returned list must be passed to restoreArraySpeciesBatch
      720  +     // in a `finally` block.
      721  +     function neutralizeArraySpeciesBatch(context, args) {
      722  +         const saved = [];
      723  +         const c = neutralizeArraySpeciesOn(context);
      724  +         if (c) saved[saved.length] = c;
      725  +         if (args) {
      726  +             // SECURITY: args is @other(safe-array) produced by otherFromThisArguments,
      727  +             // length/index access is safe. We defensively use a cached length to
      728  +             // avoid accidental getter invocation.
      729  +             const len = args.length | 0;
      730  +             for (let i = 0; i < len; i++) {
      731  +                 const s = neutralizeArraySpeciesOn(args[i]);
      732  +                 if (s) saved[saved.length] = s;
      733  +             }
      734  +         }
      735  +         return saved;
      736  +     }
      737  +
      738  +     // SECURITY: Restore every host array in a saved list. Must not throw.
      739  +     function restoreArraySpeciesBatch(savedList) {
      740  +         if (!savedList) return;
      741  +         const len = savedList.length | 0;
      742  +         for (let i = 0; i < len; i++) {
      743  +             restoreArraySpeciesOn(savedList[i]);
      744  +         }
      745  +     }
      746  +
  605 747          function thisDefaultGet(handler, object, key, desc) {
  606 748              // Note: object@other(unsafe) key@prim desc@other(safe)
  607 749              let ret; // @other(unsafe)
```

```
@@ -656,6 +798,36 @@ function createBridge(otherInit, registerProxy) {
656  798                 const object = getHandlerObject(this); // @other(unsafe)
657  799                 switch (key) {
658  800                     case 'constructor': {
     801 +                       // SECURITY (GHSA-grj5-jjm8-h35p): If the underlying object is a
     802 +                       // host-realm array, ALWAYS return the sandbox-realm Array constructor
     803 +                       // (via the target's prototype). This neutralizes the species channel
     804 +                       // that V8's ArraySpeciesCreate reads via `this.constructor` during
     805 +                       // Array.prototype.{map,filter,slice,concat,splice,...}. Attackers
     806 +                       // who install a malicious `constructor` (directly, via defineProperty,
     807 +                       // via Object.assign, or via prototype injection) cannot leak it back
     808 +                       // into V8's species resolution, because this trap short-circuits to
     809 +                       // the safe sandbox Array. Sandbox-originated arrays are unaffected
     810 +                       // -- their target prototype is sandbox Array.prototype, and its
     811 +                       // .constructor is sandbox Array (the same return value).
     812 +                       //
     813 +                       // This covers the case where sandbox Array.prototype.map runs on
     814 +                       // a host-backed proxy and reads `r.constructor` via this proxy.get
     815 +                       // trap. The apply-trap-based neutralize (below) covers the case
     816 +                       // where a host Array.prototype.map runs in the host realm on the
     817 +                       // raw array directly (via otherReflectApply).
     818 +                       let isArr = false;
     819 +                       try {
     820 +                           isArr = thisArrayIsArray(target);
     821 +                       } catch (e) {}
```

```
      822 +                     if (isArr) {
      823 +                         // SECURITY: return the cached this-realm Array constructor.
      824 +                         // Do NOT read via `proto.constructor` -- that's vulnerable to
      825 +                         // prototype pollution (Array.prototype.constructor = attackerFn).
      826 +                         // thisArrayCtor is captured at module load time before any
      827 +                         // sandbox code can execute, so it is immutable from the
      828 +                         // attacker's perspective.
      829 +                         return thisArrayCtor;
      830 +                     }
659   831                       const desc = otherSafeGetOwnPropertyDescriptor(object, key);
660   832                       if (desc) {
661   833                           if (desc.value && isDangerousFunctionConstructor(desc.value)) return {};
          @@ -739,19 +911,27 @@ function createBridge(otherInit, registerProxy) {
739   911                 // Note: target@this(unsafe) context@this(unsafe) args@this(safe-array) throws@this(unsafe)
740   912                 const object = getHandlerObject(this); // @other(unsafe)
741   913                 let ret; // @other(unsafe)
      914 +               let savedSpecies = null; // SECURITY: GHSA-grj5-jjm8-h35p -- see neutralizeArraySpeciesBatch
742   915                 try {
743   916                     context = otherFromThis(context);
744   917                     args = otherFromThisArguments(args);
745       -                 // Neutralize Array species before the host call.
746       -                 // V8's ArraySpeciesCreate reads constructor[Symbol.species] on raw host
747       -                 // arrays, bypassing proxy traps. Setting constructor = undefined forces
748       -                 // the default Array constructor, which is safe.
749       -                 neutralizeArraySpeciesArgs(context, args);
      918 +                 // SECURITY (GHSA-grj5-jjm8-h35p): Before invoking the host function,
```

```
                +                // neutralize any attacker-installed `constructor`/Symbol.species channel
                +                // on every host-realm array reachable as `context` or as a top-level
                +                // argument. V8's ArraySpeciesCreate reads these properties on the raw
                +                // object and will store raw host values into a sandbox-visible array,
                +                // bypassing the bridge, unless we shadow them here. This batch+restore
                +                // design supersedes the no-restore neutralize from #563 — that variant
                +                // permanently mutated host arrays' `constructor` to undefined, which
                +                // breaks legitimate downstream reads of `arr.constructor`.
                +                savedSpecies = neutralizeArraySpeciesBatch(context, args);
750  928                         ret = otherReflectApply(object, context, args);
751       -                // Re-neutralize after the call in case the host function restored constructor.
752       -                neutralizeArraySpeciesArgs(context, args);
753  929                 } catch (e) { // @other(unsafe)
754  930                     throw thisFromOtherForThrow(e);
     931 +               } finally {
     932 +                   // SECURITY: restore the pre-call species state even if the host call
     933 +                   // threw. `restoreArraySpeciesBatch` is guaranteed not to throw.
     934 +                   restoreArraySpeciesBatch(savedSpecies);
755  935                 }
756  936                 return thisFromOther(ret);
757  937             }
                @@ -760,11 +940,19 @@ function createBridge(otherInit, registerProxy) {
760  940                 // Note: target@this(unsafe) args@this(safe-array) newTarget@this(unsafe) throws@this(unsafe)
761  941                 const object = getHandlerObject(this); // @other(unsafe)
762  942                 let ret; // @other(unsafe)
     943 +               let savedSpecies = null; // SECURITY: GHSA-grj5-jjm8-h35p
763  944                 try {
764  945                     args = otherFromThisArguments(args);
```

```
                946 +                 // SECURITY (GHSA-grj5-jjm8-h35p): constructors can internally invoke
                947 +                 // ArraySpeciesCreate on argument arrays (e.g., Array(arr) copies, typed
                948 +                 // array constructors, Promise.all on an iterable), so neutralize args
                949 +                 // before the call as well.
                950 +                 savedSpecies = neutralizeArraySpeciesBatch(null, args);
765             951                   ret = otherReflectConstruct(object, args);
766             952               } catch (e) { // @other(unsafe)
767             953                   throw thisFromOtherForThrow(e);
                954 +             } finally {
                955 +                 restoreArraySpeciesBatch(savedSpecies);
768             956               }
769             957               return thisFromOtherWithFactory(getHandlerFactory(this), ret, thisFromOther(object));
770             958           }
```

test/ghsa/GHSA-grj5-jjm8-h35p/repro.js

```
                @@ -0,0 +1,244 @@
                1 + /**
                2 +  * GHSA-grj5-jjm8-h35p — Array species self-return sandbox escape
                3 +  *
                4 +  *
                5 +  * ## Vulnerability
                6 +  * V8's `ArraySpeciesCreate` (invoked internally by `Array.prototype.{map,filter,
                7 +  * slice,concat,splice,flat,flatMap}`) reads `this.constructor[Symbol.species]`
                8 +  * directly on raw objects, bypassing the bridge proxy's trap handlers. An
                9 +  * attacker walked the prototype chain with `({}).__lookupGetter__` + `Buffer.apply`
               10 +  * to obtain host `Object`, used `ho.entries({})` to get a host-realm array `r`,
               11 +  * installed a sandbox `constructor` whose `[Symbol.species]` returned `r` itself,
               12 +  * then called `r.map(f)` — causing V8 to write raw host values directly into the
               13 +  * sandbox-visible array and bypass bridge sanitisation entirely. Cross-reference:
```

```
14 + * docs/ATTACKS.md Category 18 (Array Species Self-Return via Constructor
15 + * Manipulation).
16 + *
17 + * ## Fix
18 + * Two-layer defense in `lib/bridge.js`: (1) the proxy `get` trap for
19 + * `constructor` on host arrays now returns a cached `Array` constructor
   captured
20 + * at module load (prototype-pollution-proof), neutralising any sandbox-side
   read
21 + * of the attacker-installed constructor; (2) the `apply`/`construct` traps
22 + * neutralise every host array reachable as context or argument by shadowing
23 + * `constructor = undefined` as a data own property before `otherReflectApply`,
24 + * then restoring the prior descriptor in a `finally`. Non-configurable or
25 + * non-extensible arrays with attacker state are rejected with `VMError`.
26 + */
27 +
28 + 'use strict';
29 +
30 + const assert = require('assert');
31 + const { VM } = require('../../../lib/main.js');
32 +
33 + // SECURITY: this prelude is common across variant tests. It uses the
34 + // `({}).__lookupGetter__` + `Buffer.apply` trick to walk the prototype chain
   until
35 + // it obtains a reference to the HOST `Object` constructor. `ho.entries({})`
   then
36 + // returns a host-realm array -- the primitive surface for the species attack.
37 + const PRELUDE = `
38 +     const g = ({}).__lookupGetter__;
39 +     const a = Buffer.apply;
40 +     const p = a.apply(g, [Buffer, ['__proto__']]);
41 +     const op = p.call(p.call(p.call(p.call(Buffer.of()))));
42 +     const ho = op.constructor; // host Object
43 + `;
44 +
45 + function assertBlocked(label, code) {
46 +     const vm = new VM();
47 +     let result;
48 +     let thrown = null;
49 +     try {
```

```
50 +         result = vm.run(PRELUDE + code);
51 +     } catch (e) {
52 +         thrown = e;
53 +     }
54 +     // Test code signals the outcome through its own return value:
55 +     //   - 'BLOCKED' means the attack attempt threw inside the sandbox (caught by test code)
56 +     //   - 'NO_ESCAPE' means the attack completed but did NOT escape (fix worked silently)
57 +     // Any other string means the exploit may have succeeded.
58 +     if (thrown) return; // a bridge-level throw also counts as blocked
59 +     assert.ok(
60 +         result === 'BLOCKED' || result === 'NO_ESCAPE',
61 +         `[${label}] expected BLOCKED or NO_ESCAPE, got: ${JSON.stringify(result)}`,
62 +     );
63 + }
64 +
65 + describe('GHSA-grj5-jjm8-h35p (array species self-return escape)', () => {
66 +     it('blocks the raw species primitive (r.constructor = x + r.map writes into r)', () => {
67 +         assertBlocked(
68 +             'species-primitive',
69 +             `
70 +             try {
71 +                 const r = ho.entries({});
72 +                 r.push(1, 2);
73 +                 function x() { return r; }
74 +                 x[Symbol.species] = x;
75 +                 r.constructor = x;
76 +                 const mapped = r.map(function (v) { return 'm' + v; });
77 +                 // The defense is successful iff V8 does NOT return r as the mapped
78 +                 // array (because r.constructor is neutralized to undefined during
79 +                 // the map call, so ArraySpeciesCreate falls back to default %Array%).
80 +                 if (mapped === r) 'ESCAPE-species-returned-self';
81 +                 else if (r[0] !== 1) 'ESCAPE-r-was-mutated';
82 +                 else 'NO_ESCAPE';
```

```
83  +             } catch (e) { 'BLOCKED' }
84  +         `,
85  +         );
86  +     });
87  +
88  +     it('blocks the canonical PoC chain (cwu helper + Function extraction)', () => {
89  +         assertBlocked(
90  +             'canonical-poc',
91  +             `
92  +             try {
93  +                 function cwu(func, thiz, args) {
94  +                     const r = ho.entries({});
95  +                     args.unshift(thiz);
96  +                     const f = a.apply(a.bind, [func, args]);
97  +                     r[0] = 0;
98  +                     function x() { return r; }
99  +                     x[Symbol.species] = x;
100 +                     r.constructor = x;
101 +                     r.map(f);
102 +                     r.constructor = undefined;
103 +                     return r;
104 +                 }
105 +                 const fp = cwu(p.call, p, [a]);
106 +                 const cd = cwu(ho.getOwnPropertyDescriptors, undefined, fp);
107 +                 const xx = cwu(ho.entries, undefined, cd);
108 +                 xx[1] = [4];
109 +                 const ee = cwu(a, xx.at, xx);
110 +                 ee[1] = [1];
111 +                 const y = cwu(a, ee.at, ee);
112 +                 const ey = cwu(ho.entries, 0, y);
113 +                 ey[1] = [0];
114 +                 const ea = cwu(a, ee.at, ey);
115 +                 ea[1] = [1];
116 +                 const e = cwu(a, ee.at, ea);
117 +                 e.push([undefined,['return process']]);
118 +                 const leaked = a.apply(a, e);
119 +                 // If we got here, the fix failed and attacker has Function ctor.
120 +                 'ESCAPE-canonical:' + typeof leaked;
```

```
121 +             } catch (e) { 'BLOCKED' }
122 +         `,
123 +         );
124 +     });
125 +
126 +     it('blocks the Object.assign backdoor for installing constructor', () => {
127 +         assertBlocked(
128 +             'object-assign',
129 +             `
130 +             try {
131 +                 const r = ho.entries({});
132 +                 r.push(1);
133 +                 function x() { return r; }
134 +                 x[Symbol.species] = x;
135 +                 ho.assign(r, { constructor: x });
136 +                 const mapped = r.map(function (v) { return 'm' + v; });
137 +                 if (mapped === r) 'ESCAPE-assign-species-returned-self';
138 +                 else 'NO_ESCAPE';
139 +             } catch (e) { 'BLOCKED' }
140 +         `,
141 +         );
142 +     });
143 +
144 +     it('rejects non-configurable attacker-installed constructor', () => {
145 +         assertBlocked(
146 +             'non-configurable',
147 +             `
148 +             try {
149 +                 const r = ho.entries({});
150 +                 r.push(1);
151 +                 function x() { return r; }
152 +                 x[Symbol.species] = x;
153 +                 ho.defineProperty(r, 'constructor', {
154 +                     value: x, writable: false, enumerable: false, configurable:
    false
155 +                 });
156 +                 // After this, ANY host call that includes r as context/arg
    must be rejected
157 +                 // because we cannot safely neutralize the species channel.
158 +                 const mapped = r.map(function (v) { return 'm' + v; });
```

```
159 +                 if (mapped === r) 'ESCAPE-nonconfig-species';
160 +                 else 'NO_ESCAPE';
161 +             } catch (e) { 'BLOCKED' }
162 +         `,
163 +         );
164 +     });
165 +
166 +     it('blocks prototype-level constructor injection (intermediate proto)', ()
    => {
167 +         assertBlocked(
168 +             'proto-level',
169 +             `
170 +             try {
171 +                 const r = ho.entries({});
172 +                 r.push(1);
173 +                 function x() { return r; }
174 +                 x[Symbol.species] = x;
175 +                 const fake = ho.create(Array.prototype);
176 +                 fake.constructor = x;
177 +                 ho.setPrototypeOf(r, fake);
178 +                 const mapped = r.map(function (v) { return 'm' + v; });
179 +                 if (mapped === r) 'ESCAPE-proto-species';
180 +                 else 'NO_ESCAPE';
181 +             } catch (e) { 'BLOCKED' }
182 +         `,
183 +         );
184 +     });
185 +
186 +     it('rejects preventExtensions-d host arrays with attacker state', () => {
187 +         assertBlocked(
188 +             'prevent-extensions',
189 +             `
190 +             try {
191 +                 const r = ho.entries({});
192 +                 r.push(1);
193 +                 function x() { return r; }
194 +                 x[Symbol.species] = x;
195 +                 // Install constructor first (as configurable), then
    preventExtensions.
```

```
196 +                // After preventExtensions, existing own properties can be reconfigured
197 +                // (as long as they stay configurable), so the fix still neutralizes.
198 +                r.constructor = x;
199 +                ho.preventExtensions(r);
200 +                const mapped = r.map(function (v) { return 'm' + v; });
201 +                if (mapped === r) 'ESCAPE-preventExt-species';
202 +                else 'NO_ESCAPE';
203 +            } catch (e) { 'BLOCKED' }
204 +        `,
205 +        );
206 +    });
207 +
208 +    it('blocks species attacks through filter and slice', () => {
209 +        assertBlocked(
210 +            'filter-slice',
211 +            `
212 +            try {
213 +                const r = ho.entries({});
214 +                r.push(1, 2);
215 +                function x() { return r; }
216 +                x[Symbol.species] = x;
217 +                r.constructor = x;
218 +                const filtered = r.filter(function () { return true; });
219 +                const sliced = r.slice(0);
220 +                if (filtered === r || sliced === r) 'ESCAPE-filter-slice-species';
221 +                else 'NO_ESCAPE';
222 +            } catch (e) { 'BLOCKED' }
223 +        `,
224 +        );
225 +    });
226 +
227 +    it('blocks species attack on concat result', () => {
228 +        assertBlocked(
229 +            'concat',
230 +            `
231 +            try {
232 +                const r = ho.entries({});
```

```
233 +                 r.push(1);
234 +                 function x() { return r; }
235 +                 x[Symbol.species] = x;
236 +                 r.constructor = x;
237 +                 const concated = r.concat([2]);
238 +                 if (concated === r) 'ESCAPE-concat-species';
239 +                 else 'NO_ESCAPE';
240 +             } catch (e) { 'BLOCKED' }
241 +         `,
242 +         );
243 +     });
244 + });
```

Comments 0

Please sign in to comment.