جاوااسکریپت برای باگ‌بانتی

آموزش مباحث پیش‌نیاز، همراه با تمرین و پاسخ تشریحی

از مبانی زبان تا DOM، سیاست هم‌مبدأ، CORS، خواندن کدهای مینیفای‌شده و آسیب‌پذیری‌های رایج سمت کلاینت

فهرست و نقشه راه

این جزوه مجموعه‌ای از مباحث جاوااسکریپت است که برای شکار باگ در برنامه‌های وب واقعاً به آن‌ها نیاز دارید. هدف، یادگیری «کل» جاوااسکریپت نیست؛ هدف این است که بتوانید کد سمت کلاینت را **بخوانید**، **بفهمید** و **نقطه‌های ضعفش** را تشخیص دهید.

1. مبانی زبان: متغیرها، نوع‌ها، مقایسه، تابع، scope و closure
2. داده‌ها: آرایه، آبجکت، JSON، رشته‌ها، URL و کدگذاری‌ها
3. prototype و `this` (پایه‌ی Prototype Pollution)
4. برنامه‌نویسی ناهمگام: async/await، Promise، event loop و fetch
5. DOM و رویدادها؛ مفهوم Source و Sink
6. مدل امنیتی مرورگر: SOP، کوکی، postMessage، CSP، CORS
7. خواندن و تحلیل جاوااسکریپت سایت هدف (Recon سمت کلاینت)
8. عبارت باقاعده (Regex)
9. Node.js برای ساخت ابزار کوچک
10. آسیب‌پذیری‌های مرتبط: CORS، Prototype Pollution، Open Redirect، XSS
11. تمرین‌ها (۳۰ تمرین)
12. پاسخ تمرین‌ها
13. مسیر ادامه و منابع تمرین

> **پیشنهاد مطالعه:** هر کد را در کنسول مرورگر (کلید F12) یا با دستور `node file.js` اجرا کنید. فقط خواندن کافی نیست؛ جاوااسکریپت را باید با دست نوشت.

> **یادآوری اخلاقی و قانونی:** تکنیک‌های این جزوه را فقط روی برنامه‌هایی تست کنید که در محدوده (Scope) یک برنامه‌ی باگ‌بانتی مجاز هستند، یا روی آزمایشگاه‌های شخصی خودتان (مثل OWASP Juice Shop که محلی نصب می‌شود). تست بدون مجوز غیرقانونی است.

فصل ۱: مبانی زبان

۱.۱ متغیرها و scope

سه روش تعریف متغیر وجود دارد: `var` (قدیمی، با scope تابعی)، `let` و `const` (جدید، با scope بلاکی). در کدهای قدیمی سایت‌ها زیاد `var` می‌بینید.

```
var a = 1;        // scope تابعی ، hoist می‌شود
let b = 2;        // scope بلاکی، قابل تغییر
const c = 3;      // scope بلاکی، قابل جایگزینی نیست

console.log(a);   // 1
if (true) {
  var x = 10;
  let y = 20;
}
console.log(x);   // 10  (var از بلاک بیرون می‌آید)
console.log(y);   // ReferenceError
```

Hoisting: تعریف `var` و توابع به بالای scope «کشیده» می‌شوند. `var` مقدار اولیه‌ی `undefined` می‌گیرد، اما `let` / `const` قبل از تعریف در «ناحیه‌ی مرده‌ی زمانی» (TDZ) هستند و خطا می‌دهند.

۱.۲ نوع‌های داده و تبدیل نوع

نوع‌های ابتدایی: `string`، `number`، `boolean`، `null`، `undefined`، `symbol`، `bigint`. بقیه‌چیز (آرایه، تابع، تاریخ و ...) `object` است.

```
typeof "hi"        // "string"
typeof 42          // "number"
typeof null        // "object"   (یک اشتباه تاریخی زبان)
typeof []          // "object"
Array.isArray([])  // true

"5" + 1            // "51"  (الحاق رشته)
"5" - 1            // 4     (تبدیل به عدد)
true + 1           // 2
```

مقایسه: `==` قبل از مقایسه نوع‌ها را تبدیل می‌کند، اما `===` نه. به همین دلیل بسیاری از باگ‌های منطقی در بررسی‌های احراز هویت و مجوز به `==` برمی‌گردد.

```
0 == "0"           // true
0 == ""            // true
"" == "0"          // false   (مقایسه‌ی تعدی‌پذیر نیست!)
null == undefined  // true
null === undefined // false
NaN == NaN         // false   (برای تشخیص: Number.isNaN)
```

Truthy / Falsy: مقادیر falsy فقط این‌ها هستند: `false`، `0`، `-0`، `0n`، `""`، `null`، `undefined`، `NaN`. بقیه truthy هستند (حتی `"0"` و `[]` و `{}`).

> **نکته‌ی امنیتی:** اگر سرور یا کلاینت ورودی را بدون بررسی نوع بپذیرد (مثلاً انتظار رشته دارد ولی آرایه یا آبجکت می‌رسد)، رفتار غیرمنتظره رخ می‌دهد. به این خطا Type Confusion می‌گویند.

۱.۳ توابع، callback و closure

```
function add(a, b) { return a + b; }          // تعریف تابع
const add2 = function (a, b) { return a + b; }; // تابع به‌صورت عبارت
const add3 = (a, b) => a + b;                  // arrow function

// callback: تابعی که به تابع دیگر داده می‌شود
[1, 2, 3].forEach(function (n) { console.log(n); });

// IIFE: (در کدهای مینیفای‌شده بسیار رایج) تابعی که بلافاصله اجرا می‌شود
(function () {
  var secret = "inside";
  console.log(secret);
})();

// closure: تابع داخلی به متغیرهای تابع بیرونی دسترسی دارد
function counter() {
  let n = 0;
  return function () { return ++n; };
}
const next = counter();
next(); // 1
next(); // 2
```

۱.۴ شرط‌ها و حلقه‌ها

```
for (let i = 0; i < 3; i++) {}
for (const v of [10, 20]) {}          // روی مقادیر
for (const k in {a: 1, b: 2}) {}      // روی کلیدها (از جمله کلیدهای ارثی شامل prototype!)
while (cond) {}
switch (x) { case 1: break; default: }
const msg = `سلام ${name}`;           // template literal
```

یادتان باشد `for...in` کلیدهای زنجیره‌ی prototype را هم می‌شمارد؛ این رفتار در فصل prototype pollution مهم می‌شود.

فصل ۲: داده‌ها

۲.۱ آرایه‌ها

```
const nums = [3, 8, 1, 9, 4];
nums.map(n => n * 2);              // [6, 16, 2, 18, 8]
nums.filter(n => n > 3);            // [8, 9, 4]
nums.find(n => n > 5);             // 8
nums.includes(9);                   // true
nums.reduce((s, n) => s + n, 0);   // 25
"a,b,c".split(",");                 // ["a","b","c"]
["a","b"].join("-");                // "a-b"
```

۲.۲ آبجکت‌ها

```
const user = { name: "Sara", role: "user", address: { city: "Tehran" } };
user.name;                 // دسترسی با نقطه
user["role"];              // دسترسی با براکت (کلید دلخواه و پویا)
Object.keys(user);         // ["name","role","address"]
Object.entries(user);      // [["name","Sara"], ...]
"role" in user;            // true  (شامل prototype هم می‌شود)
Object.hasOwn(user, "role"); // true (فقط ویژگی‌های خود آبجکت)

const { name, address: { city } } = user;   // destructuring
const copy = { ...user, role: "admin" };     // spread
user.profile?.age;                           // optional chaining، بدون خطا
```

۲.۳ JSON

```
const text = '{"id": 7, "tags": ["a","b"]}';
const obj = JSON.parse(text);       // رشته به آبجکت (ممکن است خطا بدهد)
JSON.stringify(obj);                // آبجکت به رشته
```

تقریباً تمام ارتباط‌های مرورگر و سرور (API) با JSON انجام می‌شود. وقتی در Burp یا DevTools پاسخ یک API را می‌بینید، باید JSON را سریع بخوانید.

۲.۴ رشته‌ها و کدگذاری‌ها

در باگ‌بانتی با انواع کدگذاری سروکار دارید: Unicode، Base64، URL-encoding و Hex. جاوااسکریپت همه را دارد:

```
encodeURIComponent("a b&c=d");   // "a%20b%26c%3Dd"
decodeURIComponent("%3Cb%3E");     // "<b>"
btoa("admin");                     // "YWRtaW4="
atob("YWRtaW4=");                  // "admin"
"A".charCodeAt(0);                 // 65
String.fromCharCode(72, 105);      // "Hi"
"\x68\x65\x6c\x6c\x6f";            // "hello"  (escape هگز)
"\u0068\u0069";                    // "hi"     (escape یونیکد)
```

مبهم‌سازی کد (Obfuscation) و دور زدن فیلترها اغلب از همین escapeها استفاده می‌کند؛ پس باید بتوانید آن‌ها را بخوانید.

۲.۵ URL و پارامترها

```
const u = new URL("https://example.com/search?q=test&page=2#top");
u.hostname;                // "example.com"
u.pathname;                // "/search"
u.searchParams.get("q");   // "test"   (همیشه رشته!)
u.hash;                    // "#top"

new URLSearchParams(location.search).get("next");
```

تقریباً هر «Source» ورودی در برنامه‌های وب (query string، hash، path) از همین طریق خوانده می‌شود.

فصل ۳: prototype و `this`

۳.۱ زنجیره‌ی prototype

هر آبجکت در جاوااسکریپت یک پیوند پنهان به آبجکت دیگری دارد (prototype). وقتی ویژگی‌ای در خود آبجکت نباشد، موتور زبان در زنجیره‌ی prototype دنبالش می‌گردد. اغلب این زنجیره به `Object.prototype` ختم می‌شود.

```
const o = {};
o.toString;                              // ارث رسیده Object.prototype تا بعی که از
o.__proto__ === Object.prototype;        // true
Object.getPrototypeOf(o) === Object.prototype; // true

function Dog(name) { this.name = name; }
Dog.prototype.bark = function () { return this.name + " woof"; };
new Dog("Rex").bark();                   // "Rex woof"
```

۳.۲ مقدار `this`

نحوه‌ی فراخوانی	مقدار this
روش آبجکت: `obj.f()`	همان `obj`
تابع ساده: `f()`	آبجکت سراسری یا `undefined` (در strict mode)
arrow function	this را از scope بیرونی «قرض می‌گیرد»
`f.call(x)` / `f.bind(x)`	دقیقاً همان `x`
`new F()`	آبجکت تازه‌ساخته‌شده

۳.۳ Prototype Pollution: ایده‌ی اصلی

اگر مهاجم بتواند ویژگی‌ای روی `Object.prototype` بگذارد، آن ویژگی روی «همه‌ی» آبجکت‌های برنامه ظاهر می‌شود. معمول‌ترین علت، توابع ادغام (merge / extend / deepCopy) ناامن است:

```
function merge(target, source) {
  for (const key in source) {
    if (typeof source[key] === "object" && source[key] !== null) {
      target[key] = target[key] || {};
      merge(target[key], source[key]);
    } else {
      target[key] = source[key];
    }
  }
  return target;
}

merge({}, JSON.parse('{"__proto__": {"isAdmin": true}}'));
console.log(({}).isAdmin);   // true  !!!
```

چرا؟ `JSON.parse` کلید `__proto__` را مانند یک کلید معمولی می‌سازد، و حلقه‌ی `merge` وقتی `target["__proto__"]` را می‌خواند، به `Object.prototype` می‌رسد و `isAdmin` را روی آن می‌نویسد.

پیشگیری: رد کردن کلیدهای `__proto__`، `constructor`، `prototype`؛ استفاده از `Object.create(null)` یا `Map`؛ به‌روزرسانی کتابخانه‌های آسیب‌پذیر.

فصل ۴: برنامه‌نویسی ناهمگام

۴.۱ Event loop

جاوااسکریپت تک‌نخی است. کارهای زمان‌بر (شبکه، تایمر) به صف سپرده می‌شوند و بعد از خالی شدن «پشته»ی فراخوانی اجرا می‌شوند. ترتیب اولویت: ۱) کد همگام، ۲) صف microtask (مثل `.then`)، ۳) صف macrotask (مثل `setTimeout`).

۴.۲ Promise و async/await

```
fetch("/api/me")
  .then(r => r.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));

// همان کد با async/await
async function me() {
  try {
    const r = await fetch("/api/me", { credentials: "include" });
    if (!r.ok) throw new Error("HTTP " + r.status);
    return await r.json();
  } catch (e) {
    console.error(e);
  }
}

await Promise.all([fetch("/a"), fetch("/b")]);     // همه‌ی موفق، یا اولین خطا
await Promise.allSettled([fetch("/a"), fetch("/b")]); // نتیجه‌ی همه، موفق یا ناموفق
```

۴.۳ ارسال درخواست با fetch

```
fetch("/api/profile", {
  method: "POST",
  credentials: "include",                  // کوکی‌ها هم فرستاده شوند
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ name: "Sara" })
});
```

این‌که کوکی فرستاده شود یا نه، و این‌که مرورگر اجازه‌ی خواندن پاسخ بین مبدأهای مختلف را بدهد یا نه، به فصل بعدی (SOP و CORS) مربوط می‌شود.

فصل ۵: DOM و رویدادها

۵.۱ کار با DOM

```
const el = document.getElementById("msg");
document.querySelector(".card");
document.querySelectorAll("a[href]");

el.textContent = "<b>hi</b>";   // تفسیر نمی‌شود (امن) HTML متن خام؛
el.innerHTML   = "<b>hi</b>";   // تفسیر می‌شود (خطرناک با ورودی کاربر) به‌صورت HTML

const a = document.createElement("a");
a.href = "/home"; a.textContent = "Home";
document.body.appendChild(a);

el.addEventListener("click", (event) => { console.log(event.target); });
```

۵.۲ اشیای مرورگر مهم

شیء	کاربرد / اهمیت
`location.href / search / hash`	آدرس فعلی؛ منبع رایج ورودی کنترل‌شده توسط مهاجم
`document.cookie`	کوکی‌های بدون HttpOnly
`document.referrer`	صفحه‌ی ارجاع‌دهنده
`window.name`	رشته‌ای که بین صفحه‌ها (حتی بین مبدأها) باقی می‌ماند
`localStorage / sessionStorage`	ذخیره‌ی داده؛ توکن‌ها گاهی اینجا نگه داشته می‌شوند
`window.postMessage`	ارتباط بین پنجره‌ها و iframeها

۵.۳ Source و Sink

تقریباً تمام آسیب‌پذیری‌های DOM-based از یک الگو پیروی می‌کنند: داده‌ی کنترل‌شده توسط مهاجم (**Source**) بدون پاک‌سازی به یک تابع خطرناک (**Sink**) می‌رسد.

Source (ورودی)	Sink (مقصد خطرناک)

`location.search, location.hash, location.href`
`document.referrer, window.name`
`postMessage event.data, localStorage`
	`eval(), Function()`
	`setTimeout("string"), setInterval("string")`
	`element.innerHTML, outerHTML, document.write()`
	`location = x, location.href = x`
	`script.src = x, a.href = "javascript:..."`

هنگام خواندن یک فایل JS، به‌دنبال مسیر داده بگردید: «آیا مقداری از ستون Source به ستون Sink می‌رسد؟ آیا چیزی در میانه پاکسازی می‌کند؟»

فصل ۶: مدل امنیتی مرورگر

۶.۱ سیاست هم‌مبدأ (Same-Origin Policy)

«مبدأ» (Origin) ترکیبی از سه چیز است: **scheme + host + port**. صفحه‌ای از یک مبدأ به‌طور پیش‌فرض نمی‌تواند پاسخ‌ها و DOM مبدأ دیگر را بخواند.

مقایسه با مبدأ https://app.example.com	هم‌مبدأ؟
`https://app.example.com/other`	بله
`http://app.example.com`	خیر (scheme)
`https://api.example.com`	خیر (host)
`https://app.example.com:8443`	خیر (port)

SOP جلوی «خواندن» پاسخ را می‌گیرد، نه لزوماً جلوی «ارسال» درخواست. به همین دلیل CSRF وجود دارد.

۶.۲ کوکی‌ها

- `HttpOnly`: جاوااسکریپت نمی‌تواند کوکی را بخواند (جلوی سرقت مستقیم با XSS).
- `Secure`: فقط روی HTTPS ارسال شود.
- `SameSite=Strict/Lax/None`: تعیین می‌کند کوکی در درخواست‌های بین‌سایتی ارسال شود یا نه (دفاع در برابر CSRF).
- `Domain` و `Path`: دامنه و مسیر معتبر برای کوکی.

> حتی با `HttpOnly`، یک XSS همچنان خطرناک است: مهاجم می‌تواند از داخل صفحه‌ی قربانی درخواست‌هایی با هویت او بفرستد و پاسخ‌ها را بخواند.

۶.۳ CORS

CORS مکانیزمی است که سرور با آن به مرورگر اجازه می‌دهد SOP را برای مبدأهای مشخصی سبک کند، با هدرهایی مثل:

```
Access-Control-Allow-Origin: https://trusted.example.com
Access-Control-Allow-Credentials: true
```

پیکربندی خطرناک: وقتی سرور مقدار هدر `Origin` درخواست را عیناً در `Access-Control-Allow-Origin` بازتاب می‌دهد و همزمان `Allow-Credentials: true` دارد، هر سایتی می‌تواند با کوکی‌های قربانی، داده‌های حساس را بخواند. (مقدار `*` همراه با credentials را مرورگرها رد می‌کنند.)

CSP ۶.۴

سیاست امنیت محتوا (Content-Security-Policy) هدری است که مشخص می‌کند اسکریپت‌ها، استایل‌ها و تصاویر از کدام منابع بارگذاری شوند. مثلاً `script-src 'self'` اجرای اسکریپت‌های inline و خارجی را محدود می‌کند و بهره‌برداری از XSS را سخت‌تر می‌کند. CSP ضعیف (مثلاً با `'unsafe-inline'` یا دامنه‌های بسیار باز) می‌تواند دور زده شود.

postMessage ۶.۵

```
// فرستنده
otherWindow.postMessage({ type: "hello" }, "https://trusted.example.com");

// گیرنده
window.addEventListener("message", (e) => {
  if (e.origin !== "https://trusted.example.com") return;   // بررسی ضروری
  console.log(e.data);
});
```

اگر گیرنده‌ی پیام `e.origin` را بررسی نکند و `e.data` را به یک Sink بدهد، آسیب‌پذیر است.

iframe و Clickjacking ۶.۶

اگر سایتی بتواند در iframe سایت دیگر نمایش داده شود، مهاجم می‌تواند صفحه‌ی شفاف رویش بگذارد و کاربر را به کلیک فریب دهد. دفاع: `X-Frame-Options` یا `frame-ancestors` در CSP.

فصل ۷: خواندن جاوااسکریپت سایت هدف

یکی از ارزشمندترین مهارت‌های هانتر وب همین است. فایل‌های JS اغلب مسیرهای پنهان API، پارامترها، و منطق بررسی دسترسی سمت کلاینت را لو می‌دهند.

۷.۱ پیدا کردن فایل‌ها

- DevTools ← تب Sources و Network (فیلتر JS)
- View Source صفحه و جست‌وجوی تگ‌های `<script src>`
- Burp Suite ← Proxy history و Sitemap
- فایل‌های Source Map (پسوند `.map` یا کامنت `//# sourceMappingURL=`) که گاهی کد اصلی خوانا را در اختیار می‌گذارند

۷.۲ خوانا کردن کد مینیفای‌شده

در DevTools روی دکمه‌ی `{ }` (Pretty print) کنار فایل کلیک کنید. ابزارهای دیگر: `prettier` و `js-beautify`.

۷.۳ چه چیزهایی را جست‌وجو کنیم؟

الگو	چرا مهم است
`fetch( / axios / XMLHttpRequest / "/api/"`	مسیرهای API، پارامترها و متدها
`isAdmin / role / permission / debug`	بررسی‌های دسترسی سمت کلاینت (قابل دور زدن اگر سرور تأیید نکند)
`innerHTML / document.write / eval(`	Sinkهای احتمالی XSS
`location.hash / location.search / postMessage`	Sourceها
`apiKey / token / secret / Bearer`	کلیدهای افشاشده (گزارش مسئولانه؛ هرگز سوءاستفاده نکنید)
`نسخه‌ی کتابخانه‌ها (jQuery, Angular, lodash ...)`	آسیب‌پذیری‌های شناخته‌شده در نسخه‌های قدیمی

۷.۴ مبهم‌سازی رایج

```
eval(atob("Y29uc29sZS5sb2coMSk="));          // کد پنهان در Base64
eval(String.fromCharCode(97,108,101,114,116)); // ساخت رشته از کد کاراکترها
"\x61\x6c\x65\x72\x74"                        // هگز escape

eval(function(p,a,c,k,e,d){...})              // packer معروف
```

احتیاط: کد ناشناس را هرگز مستقیم اجرا نکنید. برای رمزگشایی، به‌جای `eval` از `console.log` یا `atob` به‌تنهایی استفاده کنید تا فقط «نتیجه» را ببینید. ترجیحاً تحلیل را در مرورگر/ماشین مجازی جدا انجام دهید.

فصل ۸: عبارت باقاعده (Regex)

Regex برای استخراج مسیرها، توکن‌ها و الگوها از فایل‌های بزرگ JS، و برای فهمیدن فیلترهای مبتنی بر regex در برنامه‌ها ضروری است.

نماد	معنی
`. \d \w \s`	هر کاراکتر / رقم / حرف و رقم و _ / فاصله
`* + ? {n,m}`	تکرار: صفر یا بیشتر / یک یا بیشتر / اختیاری / تعداد مشخص
`[abc] [^abc]`	یکی از این‌ها / هیچ‌کدام از این‌ها
`^ $`	ابتدا و انتهای رشته
`( ) |`	گروه‌بندی و «یا»

```
const text = 'fetch("/api/v1/users"); x = "/api/v2/orders?id=1"';
const re = /["'`](\/api\/[^"'`\s]*)["'`]/g;
const found = [...text.matchAll(re)].map(m => m[1]);
console.log(found);   // ["/api/v1/users", "/api/v2/orders?id=1"]

/^\d{3}-\d{4}$/.test("123-4567");   // true
"a1b22".replace(/\d+/g, "#");       // "a#b#"
```

> عبارات باقاعده‌ی بد (مثلاً با تکرارهای تودرتو) می‌توانند باعث ReDoS شوند؛ و فیلترهای مبتنی بر regex اغلب با کدگذاری یا حالت‌های حاشیه‌ای دور زده می‌شوند.

فصل ۹: Node.js برای ساخت ابزار کوچک

Node.js امکان اجرای جاوااسکریپت خارج از مرورگر را می‌دهد. برای اسکریپت‌های کمکی (استخراج URL، ارسال درخواست، پردازش خروجی ابزارها) عالی است. نسخه‌های جدید `fetch` را به‌صورت داخلی دارند.

```
// extract-urls.js  →  node extract-urls.js app.js
const fs = require("fs");

const file = process.argv[2];
const code = fs.readFileSync(file, "utf8");

const urls = new Set(code.match(/https?:\/\/[^\s"'`<>)]+/g) || []);
for (const u of urls) console.log(u);
```

```
// status.js  →  فقط روی هدف مجاز خودتان استفاده کنید
const url = process.argv[2];
fetch(url).then(r => console.log(r.status, r.headers.get("content-type")));
```

برای مدیریت بسته‌ها: `npm install`. ابزارهای مفید برای هانتر: `prettier`، `cheerio` (تحلیل HTML)، `puppeteer` (مرورگر خودکار).

فصل ۱۰: آسیب‌پذیری‌های مرتبط با جاوااسکریپت

۱۰.۱ XSS (Cross-Site Scripting)

XSS یعنی اجرای کد جاوااسکریپت مهاجم در مبدأ برنامه‌ی قربانی. سه نوع اصلی:

نوع	توضیح
Reflected	ورودی همان درخواست، در پاسخ سرور بدون پاکسازی بازتاب می‌شود
Stored	ورودی ذخیره می‌شود (کامنت، پروفایل) و بعداً برای کاربران دیگر نمایش داده می‌شود
DOM-based	مشکل کاملاً در جاوااسکریپت کلاینت است؛ Source بدون پاکسازی به Sink می‌رسد

مثال آسیب‌پذیر DOM XSS:

```
// آسیب‌پذیر
const name = new URLSearchParams(location.search).get("name");
document.getElementById("welcome").innerHTML = "Hello " + name;

// آدرس تست (برای اثبات‌مفهوم، بی‌آزار):
// https://target.example/?name=<img src=x onerror=alert(document.domain)>

// اصلاح
document.getElementById("welcome").textContent = "Hello " + name;
```

در گزارش باگ‌بانتی از `alert(document.domain)` استفاده کنید تا ثابت شود کد در دامنه‌ی هدف اجرا شده است؛ نیازی به سرقت داده‌ی کاربران واقعی نیست و ممنوع است.

دفاع: encode خروجی بر اساس زمینه (HTML / attribute / JS / URL)، استفاده از `textContent`، کتابخانه‌ی پاک‌ساز مثل DOMPurify، و CSP قوی.

۱۰.۲ Open Redirect

```
// آسیب‌پذیر
location.href = new URLSearchParams(location.search).get("next");
// ?next=https://evil.example  →  کاربر به سایت مهاجم می‌رود

// اصلاح
function safeNext(n) {
  try {
    const u = new URL(n, location.origin);
    return u.origin === location.origin ? u.pathname + u.search : "/";
  } catch { return "/"; }
}
```

Open redirect به‌تنهایی معمولاً شدت پایین دارد، ولی در زنجیره با OAuth یا فیشینگ مهم می‌شود.

۱۰.۳ Prototype Pollution

در فصل ۳ دیدید. در سمت کلاینت، آلودگی می‌تواند به gadgetهایی در کد برنامه برسد که ویژگی‌ها را از آبجکت‌ها می‌خوانند (مثلاً `options.url`) و به Sink می‌دهند، و به XSS ختم شود. در سمت سرور (Node.js) می‌تواند به دور زدن احراز هویت یا حتی اجرای کد برسد.

۱۰.۴ مشکلات CORS و postMessage

در فصل ۶ توضیح داده شد: بازتاب Origin با credentials، و گیرنده‌ی postMessage بدون بررسی origin.

۱۰.۵ منطق سمت کلاینت و کنترل دسترسی

هر بررسی‌ای که فقط در جاوااسکریپت باشد (مثلاً مخفی کردن دکمه‌ی «ادمین» یا یک هدر `X-Role`) با دست‌کاری کلاینت دور زده می‌شود. باگ واقعی زمانی است که **سرور** همان درخواست را بدون بررسی مجوز بپذیرد (Broken Access Control / IDOR).

تمرین‌ها

ابتدا خودتان حل کنید و بعد پاسخ‌ها را ببینید. هر تمرین را حتماً در کنسول اجرا کنید.

بخش الف: مبانی (فصل ۱ و ۲)

تمرین ۱. خروجی چیست؟

```
console.log(typeof null, typeof undefined, typeof [], typeof NaN);
```

تمرین ۲. مقدار هر عبارت چیست؟

```
"5" + 1,  "5" - 1,  "5" * "2",  true + 1,  [] + [],  null + 1,  undefined + 1
```

تمرین ۳. نتیجه‌ی هر مقایسه (true / false)؟

```
0 == "0",  0 == "",  "" == "0",  null == undefined,  null === undefined,  NaN == NaN
```

تمرین ۴. خروجی (یا خطا) هر خط را بگویید و علتش را توضیح دهید:

```
console.log(a);
var a = 5;
console.log(b);
let b = 6;
```

تمرین ۵. تابع `makeCounter` را بنویسید که هر بار فراخوانی تابع بازگشتی‌اش، عدد را یکی زیاد کند و برگرداند (۱، ۲، ۳، ...). از closure استفاده کنید.

تمرین ۶. با `filter` و `map` از آرایه‌ی `[3, 8, 1, 9, 4]`، دو برابرِ عددهای بزرگ‌تر از ۳ را بسازید.

تمرین ۷. خروجی چیست؟

```
const { a, b: { c } } = { a: 1, b: { c: 2, d: 3 } };
console.log(a, c);
```

تمرین ۸. تابع `safeParse(text)` را بنویسید که JSON را پارس کند و در صورت نامعتبر بودن، به‌جای خطا `null` برگرداند.

تمرین ۹. این کدگذاری‌ها را رمزگشایی کنید (بدون اجرای کد):

```
"\x68\x65\x6c\x6c\x6f"
atob("YWRtaW4=")
String.fromCharCode(72, 105)
```

تمرین ۱۰. برای آدرس `https://example.com/search?q=test&page=2&debug=true` مقدار `page`، نوع داده‌ی آن، و hostname و pathname را با کد به‌دست آورید.

بخش ب: prototype و ناهمگام (فصل ۳ و ۴)

تمرین ۱۱. چرا `h()` نتیجه‌ی `obj.f()` را نمی‌دهد و چگونه درستش کنیم؟

```
const obj = { name: "A", f() { return this.name; } };
const h = obj.f;
console.log(obj.f(), h());
```

تمرین ۱۲. خروجی چیست و چرا؟

```
const o = {};
o.__proto__.x = 1;
const p = {};
console.log(p.x);
```

تمرین ۱۳. با تابع `merge` فصل ۳، اگر ورودی `{"__proto__": {"isAdmin": true}}` پارس و ادغام شود، چه اتفاقی می‌افتد؟ و راه‌حل چیست؟

تمرین ۱۴. ترتیب چاپ را مشخص کنید:

```
console.log(1);
setTimeout(() => console.log(2), 0);
Promise.resolve().then(() => console.log(3));
console.log(4);
```

تمرین ۱۵. تابع `getUser(id)` را با async/await بنویسید که از `/api/users/ID` با ارسال کوکی بخواند، در صورت پاسخ ناموفق خطا بدهد و در غیر این صورت JSON را برگرداند.

تمرین ۱۶. اگر یکی از Promiseهای `Promise.all` رد شود چه می‌شود؟ برای گرفتن نتیجه‌ی همه چه کنیم؟

بخش ج: DOM و امنیت مرورگر (فصل ۵ و ۶)

تمرین ۱۷. تفاوت دو خط زیر چیست؟

```
el.innerHTML   = "<img src=x onerror=alert(1)>";
el.textContent = "<img src=x onerror=alert(1)>";
```

تمرین ۱۸. هر مورد را Source یا Sink دسته‌بندی کنید: `location.hash`، `document.write`، `eval`، `document.referrer`، `innerHTML`، `window.name`، `event.data` در postMessage، `setTimeout(string)`.

تمرین ۱۹. باگ این کد چیست؟ یک آدرس تست بی‌آزار و اصلاح آن را بنویسید.

```
const params = new URLSearchParams(location.search);
document.getElementById("welcome").innerHTML = "Hello " + params.get("name");
```

تمرین ۲۰. تابع `safeLink(text, url)` بنویسید که لینک بسازد ولی فقط پروتکل‌های `http` و `https` را بپذیرد (جلوی `javascript:` را بگیرد).

تمرین ۲۱. مبدأ پایه `https://app.example.com` است. کدام آدرس‌ها هم‌مبدأ هستند؟
الف) `https://app.example.com/other` ب) `http://app.example.com/page` ج) `https://api.example.com/page` د) `https://app.example.com:8443/page` ه) `https://app.example.com.evil.com`

تمرین ۲۲. اگر کوکی نشست `HttpOnly` باشد، آیا XSS دیگر خطرناک نیست؟ توضیح دهید.

تمرین ۲۳. سرور به درخواستی با `Origin: https://evil.com` پاسخ می‌دهد: `Access-Control-Allow-Origin: https://evil.com` و `Access-Control-Allow-Credentials: true`. آیا این آسیب‌پذیری است؟ در چه شرطی؟

تمرین ۲۴. مشکل این کد چیست و چگونه اصلاح می‌شود؟

```
window.addEventListener("message", (e) => {
  document.getElementById("out").innerHTML = e.data;
});
```

بخش د: Recon و آسیب‌پذیری‌ها (فصل ۷ تا ۱۰)

تمرین ۲۵. بدون اجرای `eval` مشخص کنید این کد چه می‌کند و چگونه آن را با خیال راحت رمزگشایی می‌کنید؟

```
eval(atob("Y29uc29sZS5sb2coMSk="));
```

تمرین ۲۶. از این کد مینیفای‌شده چه اطلاعاتی برای تست امنیتی استخراج می‌کنید؟

```
var a=function(t){return fetch("/api/v2/admin/users/"+t,{headers:{"X-Role":"admin"}})};
```

تمرین ۲۷. یک regex بنویسید که همه‌ی رشته‌هایی را که با `/api/` شروع می‌شوند و داخل ' یا " یا ` هستند، از متن JS استخراج کند.

تمرین ۲۸. اسکریپت Node.js بنویسید که مسیر یک فایل JS را بگیرد و همه‌ی URLهای http/https یکتا داخلش را چاپ کند.

تمرین ۲۹. کد زیر چه مشکلی دارد و چگونه اصلاح می‌شود؟

```
location.href = new URLSearchParams(location.search).get("next");
```

تمرین ۳۰. تفاوت XSS از نوع Stored، Reflected و DOM-based را در دو سه خط بنویسید و بگویید کدام‌یک را فقط با خواندن JS کلاینت می‌توان پیدا کرد.

پاسخ تمرین‌ها

پاسخ ۱. خروجی: `object undefined object number` . `typeof null` اشتباه تاریخی زبان است؛ آرایه از نوع object است؛ `NaN` هم از نوع number است.

پاسخ ۲. `"51"` ، `4` ، `10` ، `2` ، `""` (رشته‌ی خالی)، `1` ، `NaN` . عملگر `+` اگر یک طرف رشته باشد الحاق می‌کند؛ `-` و `*` همیشه عددی هستند.

پاسخ ۳. `true` ، `true` ، `false` ، `true` ، `false` ، `false` . به‌ویژه توجه کنید `0=="0"` و `0==""` درست‌اند ولی `""=="0"` نادرست است.

پاسخ ۴. خط اول `undefined` چاپ می‌کند (`var` hoist می‌شود ولی مقدارش هنوز ندارد). خط سوم خطای `ReferenceError: Cannot access 'b' before initialization` می‌دهد، چون `let` در TDZ است.

پاسخ ۵.

```
function makeCounter() {
  let n = 0;
  return () => ++n;
}
const c = makeCounter();
c(); c(); c();   // 1, 2, 3
```

پاسخ ۶.

```
[3, 8, 1, 9, 4].filter(n => n > 3).map(n => n * 2);   // [16, 18, 8]
```

پاسخ ۷. خروجی: `1 2` . destructuring تودرتو مقدار `a` و `c` را برمی‌دارد؛ `d` نادیده گرفته می‌شود.

پاسخ ۸.

```
function safeParse(text) {
  try { return JSON.parse(text); }
  catch { return null; }
}
```

پاسخ ۹. به ترتیب: `"hello"` ، `"admin"` ، `"Hi"` .

پاسخ ۱۰.

```
const u = new URL("https://example.com/search?q=test&page=2&debug=true");
u.searchParams.get("page");         // "2"
typeof u.searchParams.get("page");  // "string"  (برای عدد: Number(...))
u.hostname;                         // "example.com"
u.pathname;                         // "/search"
```

پاسخ ۱۱. وقتی `h = obj.f` را جدا می‌کنیم، `this` دیگر `obj` نیست. `obj.f()` مقدار `"A"` را می‌دهد؛ اما `h()` به‌جای آن `undefined` (در Node)، رشته‌ی خالی (در مرورگر، چون `window.name` خوانده می‌شود)، یا خطا (در strict mode) می‌دهد. راه‌حل: `const h = obj.f.bind(obj);`.

پاسخ ۱۲. خروجی `1`. چون `o.__proto__` همان `Object.prototype` است، و نوشتن `x` روی آن روی همه‌ی آبجکت‌ها، از جمله `p`، اثر می‌گذارد. این همان Prototype Pollution است.

پاسخ ۱۳. `JSON.parse` کلید `__proto__` را به‌عنوان ویژگی معمولی می‌سازد، `for...in` آن را می‌شمارد، و `merge` به `Object.prototype` می‌رسد و `isAdmin = true` را روی آن می‌نویسد. بعد از آن `({}).isAdmin` برای تمام آبجکت‌ها `true` است. راه‌حل: نادیده گرفتن کلیدهای `__proto__`، `constructor` و `prototype`؛ استفاده از `Object.create(null)` یا `Map`؛ و به‌روز کردن کتابخانه‌ی merge.

پاسخ ۱۴. ترتیب: `1`، `4`، `3`، `2`. ابتدا کد همگام، سپس microtask (`.then`)، و در آخر macrotask (`setTimeout`).

پاسخ ۱۵.

```
async function getUser(id) {
  const r = await fetch(`/api/users/${id}`, { credentials: "include" });
  if (!r.ok) throw new Error("HTTP " + r.status);
  return r.json();
}
```

پاسخ ۱۶. `Promise.all` به‌محض اولین رد شدن، با همان خطا رد می‌شود. برای گرفتن نتیجه‌ی همه (موفق و ناموفق) از `Promise.allSettled` استفاده می‌کنیم.

پاسخ ۱۷. `innerHTML` رشته را به‌عنوان HTML تفسیر می‌کند، پس تصویر ساخته می‌شود، `onerror` اجرا می‌شود و کد اجرا می‌گردد (XSS). `textContent` همان رشته را به‌صورت متن خام نمایش می‌دهد و چیزی اجرا نمی‌شود.

پاسخ ۱۸. Source: `location.hash`، `document.referrer`، `window.name`، `event.data`. Sink: `document.write`، `eval`، `innerHTML`، `setTimeout(string)`.

پاسخ ۱۹. DOM XSS: پارامتر `name` بدون پاکسازی به `innerHTML` می‌رسد. آدرس تست: `?name=<img src=x onerror=alert(document.domain)>` اصلاح: `textContent = "Hello " + params.get("name")` (یا پاک‌سازی با DOMPurify اگر HTML لازم است).

پاسخ ۲۰.

```
function safeLink(text, url) {
  try {
    const u = new URL(url, location.origin);
    if (!["http:", "https:"].includes(u.protocol)) return null;
    const a = document.createElement("a");
    a.href = u.href;
    a.textContent = text;
    a.rel = "noopener noreferrer";
    return a;
  } catch { return null; }
}
```

پاسخ ۲۱. فقط (الف) هم‌مبدأ است. (ب) scheme متفاوت، (ج) host متفاوت، (د) port متفاوت، (ه) host کاملاً متفاوت است (`evil.com` دامنه‌ی اصلی است).

پاسخ ۲۲. خیر. `HttpOnly` فقط جلوی خواندن مستقیم کوکی را می‌گیرد. مهاجم با XSS هنوز می‌تواند از داخل صفحه‌ی قربانی، با همان نشست، درخواست بفرستد، پاسخ‌ها را بخواند، فرم‌ها را تغییر دهد، کلیدفشردن‌ها را بگیرد و ... .

پاسخ ۲۳. بله، اگر آن endpoint داده‌ی حساسِ وابسته به کاربر برگرداند (مثل پروفایل یا توکن). در این حالت JS روی `evil.com` می‌تواند با کوکی‌های قربانی درخواست بفرستد و پاسخ را بخواند. اگر endpoint فقط داده‌ی عمومی برگرداند، تأثیر امنیتی ندارد. (توجه: SameSite کوکی هم می‌تواند اثر را محدود کند.)

پاسخ ۲۴. دو مشکل: بررسی نشدن `e.origin` (هر سایتی می‌تواند پیام بفرستد) و رسیدن `e.data` به Sink به نام `innerHTML`. نتیجه DOM XSS است. اصلاح:

```
window.addEventListener("message", (e) => {
  if (e.origin !== "https://trusted.example.com") return;
  document.getElementById("out").textContent = String(e.data);
});
```

پاسخ ۲۵. رشته‌ی Base64 معادل `console.log(1)` است. برای رمزگشایی ایمن: فقط `console.log(atob("..."))` را اجرا کنید تا متن را ببینید و `eval` را حذف کنید، تا کدِ ناشناس اجرا نشود.

پاسخ ۲۶. اطلاعات: مسیر `/api/v2/admin/users/{id}` (یک endpoint مدیریتی با شناسه‌ی کاربر)، و یک هدر `X-Role: admin` که کلاینت می‌فرستد. باید در محدوده‌ی مجاز تست شود که آیا سرور نقش را از هدر ارسالی کلاینت می‌پذیرد، و آیا بدون ادمین بودن، با تغییر `id` به داده‌ی دیگران دسترسی می‌دهد (Broken Access Control / IDOR). مجوز را همیشه سرور باید بررسی کند، نه کلاینت.

پاسخ ۲۷.

```
const re = /["'`](\/api\/[^"'`\s]*)["'`]/g;
const paths = [...jsText.matchAll(re)].map(m => m[1]);
```

پاسخ ۲۸.

```
const fs = require("fs");
const code = fs.readFileSync(process.argv[2], "utf8");
const urls = new Set(code.match(/https?:\/\/[^\s"'`<>)]+/g) || []);
for (const u of urls) console.log(u);
```

پاسخ ۲۹. Open Redirect: مقدار `next` بدون اعتبارسنجی به `location.href` می‌رسد؛ مثلاً `?next=https://evil.example`. اصلاح: فقط مسیرهای هم‌مبدأ را بپذیرید یا از allowlist استفاده کنید:

```
function safeNext(n) {
  try {
    const u = new URL(n, location.origin);
    return u.origin === location.origin ? u.pathname + u.search : "/";
  } catch { return "/"; }
}
location.href = safeNext(new URLSearchParams(location.search).get("next"));
```

این تابع ورودی‌هایی مثل `//evil.com`، `https://evil.com` و `javascript:alert(1)` را به `/` برمی‌گرداند.

پاسخ ۳۰. Reflected: ورودی درخواست جاری در پاسخ سرور بازتاب می‌شود. Stored: ورودی در پایگاه‌داده ذخیره و بعداً برای دیگران نمایش داده می‌شود. DOM-based: تمام مشکل در JS کلاینت است و داده از Source به Sink می‌رسد، بدون اینکه (لزوماً) به‌صورت کنترل‌نشده از سرور برگردد. نوع DOM-based را می‌توان فقط با خواندن و تحلیل JS کلاینت پیدا کرد.

مسیر ادامه و منابع تمرین

مسیر پیشنهادی ۸ هفته‌ای

هفته	موضوع	هدف عملی
۱–۲	فصل‌های ۱ تا ۳ + تمرین‌ها	حل تمرین‌های الف و ب بدون نگاه به پاسخ
۳	فصل‌های ۴ و ۵	ساخت یک صفحه‌ی ساده که از یک API دمو داده بخواند و نمایش دهد
۴	فصل ۶	درک CORS و SOP با آزمایش دو سرور محلی روی پورت‌های مختلف
۵	فصل‌های ۷ و ۸	تحلیل فایل‌های JS یک برنامه‌ی آزمایشگاهی و استخراج endpointها
۶	فصل ۹	نوشتن یک ابزار کوچک Node.js برای استخراج مسیرها
۷–۸	فصل ۱۰	حل چالش‌های XSS و Prototype Pollution در آزمایشگاه

آزمایشگاه‌ها و منابع

- **OWASP Juice Shop**: برنامه‌ی عمداً آسیب‌پذیر که روی سیستم خودتان اجرا می‌شود (با Node.js یا Docker) و برای تمرین محلی عالی است.
- **PortSwigger Web Security Academy**: آزمایشگاه‌های رایگان XSS، DOM-based، CORS، Prototype Pollution و ...
- **Google XSS Game** و **OWASP WebGoat**: تمرین‌های XSS و مباحث پایه.
- **MDN Web Docs**: مرجع کامل جاوااسکریپت و APIهای مرورگر.
- **javascript.info**: آموزش گام‌به‌گام و عمیق زبان.
- **OWASP Cheat Sheet Series**: راهنمای دفاعی برای XSS، CSRF، CORS و CSP.

نکته‌ی پایانی: بهترین یادگیری ترکیب این سه کار است: ۱) خواندن کد واقعی، ۲) شکستن آزمایشگاه‌ها، ۳) نوشتن ابزار کوچک خودتان. همین‌که بتوانید یک فایل JS ناشناس را باز کنید و بگویید «ورودی از کجا می‌آید و به کجا می‌رسد»، از اکثر مبتدی‌ها جلوترید.