РусКрипто XXVIII НАУЧНО-ПРАКТИЧЕСКАЯ КОНФЕРЕНЦИЯ

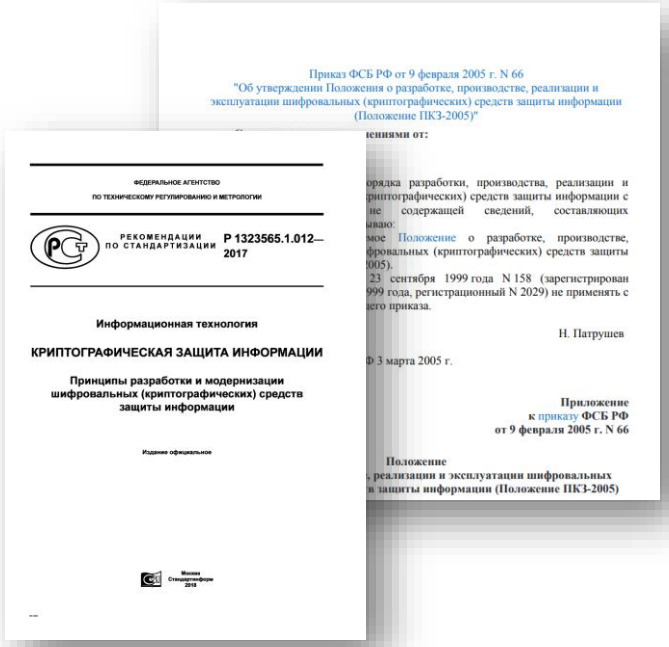
Применение задачи удовлетворения ограничений для автоматического анализа интерфейсов СКЗИ

Тырнов Филипп, инженер-аналитик, КриптоПро

План выступления

- Анализ интерфейсов СКЗИ при его сертификации;
- Интерфейсы СКЗИ;
- Методы анализа интерфейсов СКЗИ;
- Задача удовлетворения ограничений (ЗУО);
- Решатель ЗУО;
- От теории к практике;
- Итоги.

Анализ интерфейсов СКЗИ при его сертификации

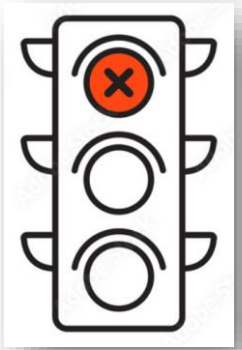


«Принципы разработки» + ПКЗ-2005

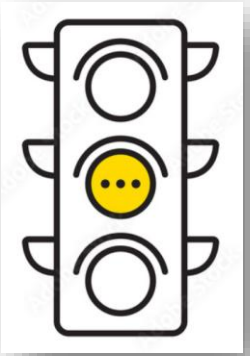
- п. 31 ПКЗ-2005: тематические исследования;
- п. 46 ПКЗ-2005: эксплуатация СКЗИ в соответствии с правилами пользования;
- п. 6.2.8 + 6.2.9. «Принципов разработки»: принимаемые функциями параметры не должны приводить к возникновению уязвимостей СКЗИ, иначе – составление соответствующего списка.

Интерфейсы СКЗИ

«Черные» функции – отсутствуют в правилах пользования



«Белые» функции – требуют оценки влияния



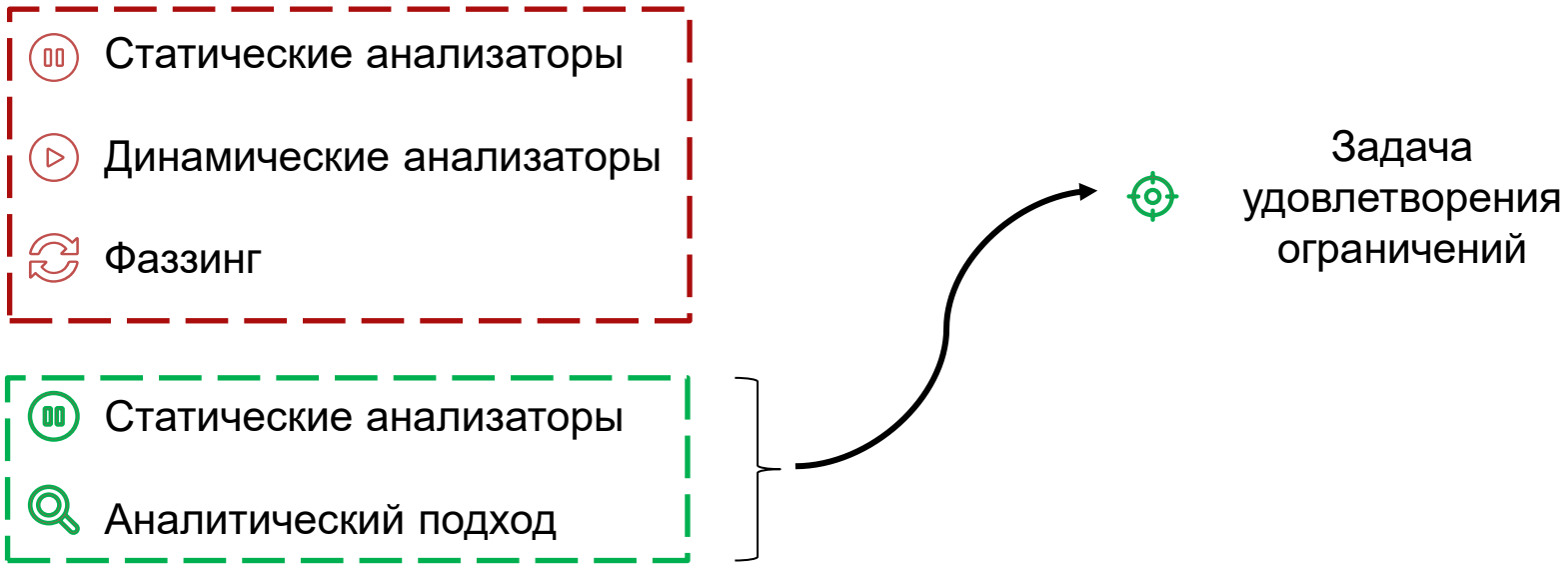
«Супер-белые» функции – не требуют исследований

Интерфейсы СКЗИ

Интерфейс CryptoAPI

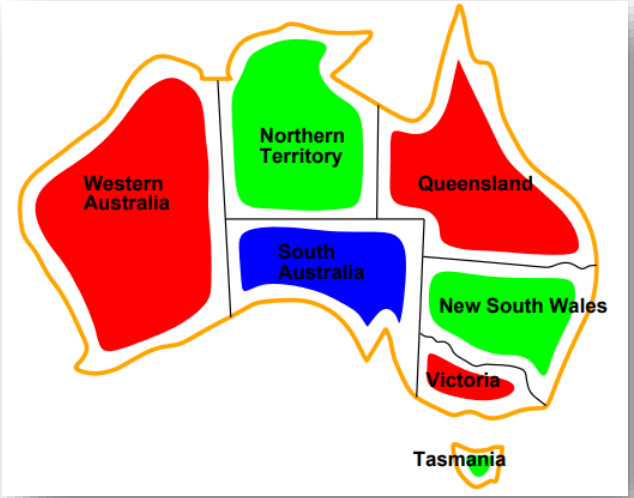
Функция	Описание	Ограничения на использование функции
Функции инициализации и настройки провайдера		
`CryptAcquireContext`	Функция создаёт дескриптор криптопровайдера с именем ключевого контейнера, определённым	Допускается использование только провайдеров из состава СКЗИ «КриптоПро CSP» (типы 75, 80 и
`CryptReleaseContext`	Функция удаляет дескриптор криптопровайдера, созданного CryptAcquireContext.	Перед вызовом данной функции все дескрипторы объектов ключей и хэширования, работа с которыми производилась совместно с удаляемым дескриптором криптопровайдера, должны быть удалены с помощью вызовов
`CryptContextAddRef`	Функция управляет счётчиком дескрипторов созданного CryptAcquireContext.	

Методы анализа интерфейса СКЗИ



Задача удовлетворения ограничений (ЗУО)

- Задается переменными, допустимыми значениями и ограничениями;
- Решение ЗУО – назначение переменным значений так, чтобы ограничения не были нарушены;
- Пример ЗУО – раскраска штатов Австралии;
- Относится к классу NP-полных задач.



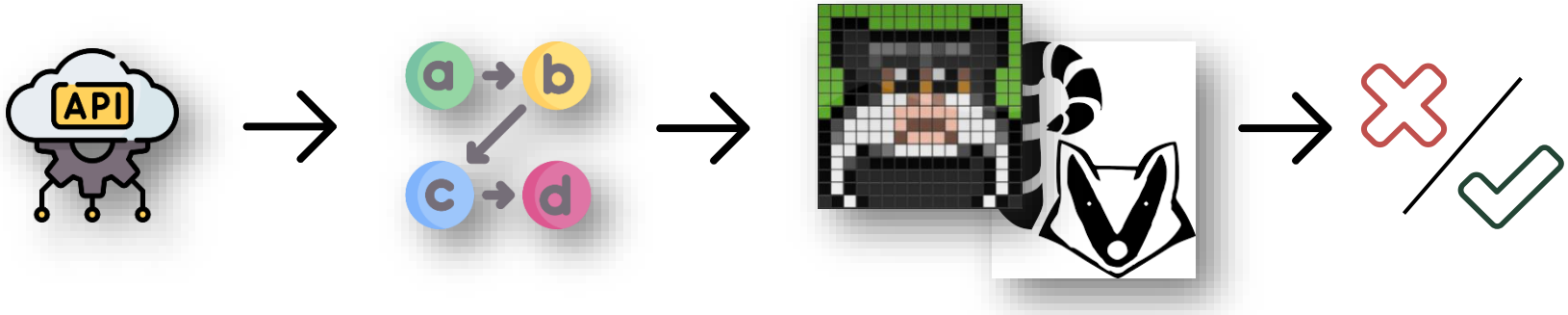
Решатель ЗУО

Tamarin Prover

- Современный стандарт для символьной верификации протоколов;
- Процесс верификации – поиск трассы вызовов, на которой нарушается ограничение;
- Успешно использовался при формальном доказательстве безопасности TLS 1.3, Apple iMessage, 5G сетей.

Решатель ЗУО – как воспользоваться в нашем случае?

- Сведение задачи анализа интерфейсов СКЗИ к ЗУО;
- Решение ЗУО – безопасна ли реализация интерфейсов или нет?
- Применение решателя ЗУО (например, Tamarin Prover / ProVerif / …);
- Раскрытие на масштабировании экспортируемых функций.



Решатель ЗУО – принцип работы

01

Кодирование параметров →

Параметры и программные интерфейсы кодируются на языке решателя ЗУО

02

Правила →

Закодированные параметры и программные интерфейсы определяют правила функционирования системы

03

Леммы →

Формирование ограничений на языке логики первого порядка

04

Решение

Решатель ЗУО осуществляет поиск трассы вызовов, на которой нарушаются ограничения

Решатель ЗУО – семантика правил и лемм

Семантика правил:

- предусловие;
- действие;
- заключение.

```
rule Test_rule:
    [предусловие]
    --[действие]->
    [заключение]
```

Семантика лемм:

- область распространения;
- определение леммы.

```
lemma Test_lemma:
    exists-trace / all-traces
    "определение леммы"
```

От теории к практике

- Рассмотрим описанный подход для API протокола TLS;
- Возьмем распространенную библиотеку – OkHttp;
- Закодируем параметры и демонстрационные функции;
- Проверим работу решателя ЗУО.

От теории к практике – уязвимая реализация

①
```
OkHttpClient client = new OkHttpClient.Builder()
    .hostnameVerifier((hostname, session) -> true)  // ← уязвимость
    .build();
```

②
```
Request request = new Request.Builder()
    .url("https://api.example.com/data")
    .addHeader("Authorization", "Bearer " + token)
    .build();
```

③
```
client.newCall(request).execute();
```

④
```
OkHttpClient client = new OkHttpClient.Builder().build();
```

От теории к практике – кодируем правила

```
rule Register_Server:
 ① [ Fr(~ltkS) ]
    -->
 ② [ !Ltk('api.example.com', ~ltkS)
 ③ , !Pk('api.example.com', pk(~ltkS)) ]
```

```
rule OkHttp_Safe_Send:
 ④ [ !Pk('api.example.com', pkS)
    , Fr(~token) ]
 ⑤ --[ SafeSend('Client', ~token, pkS) ]->
 ⑥ [ Out(aenc(<'Authorization', ~token>, pkS)) ]
```

```
rule OkHttp_Vulnerable_Send:
 ⑦ [ In(<'cert', anyHost, anyPk>)
    , Fr(~token) ]
 ⑧ --[ VulnerableSend('Client', ~token, anyPk) ]->
 ⑨ [ Out(aenc(<'Authorization', ~token>, anyPk)) ]
```

От теории к практике – кодируем леммы

```
rule OkHttp_Vulnerable_Send:
    [ In(<'cert', anyHost, anyPk>)
    , Fr(~token) ]
    --[ VulnerableSend('Client', ~token, anyPk) ]->
    [ Out(aenc(<'Authorization', ~token>, anyPk)) ]
```

```
lemma Vulnerable_Token_Is_Secret:
(1)"All token pk #i.
  (2) VulnerableSend('Client', token, pk) @ #i
      ==>
  (3) not(Ex #j. K(token) @ #j)
"
```

```
rule OkHttp_Safe_Send:
    [ !Pk('api.example.com', pkS)
    , Fr(~token) ]
    --[ SafeSend('Client', ~token, pkS) ]->
    [ Out(aenc(<'Authorization', ~token>, pkS)) ]
```

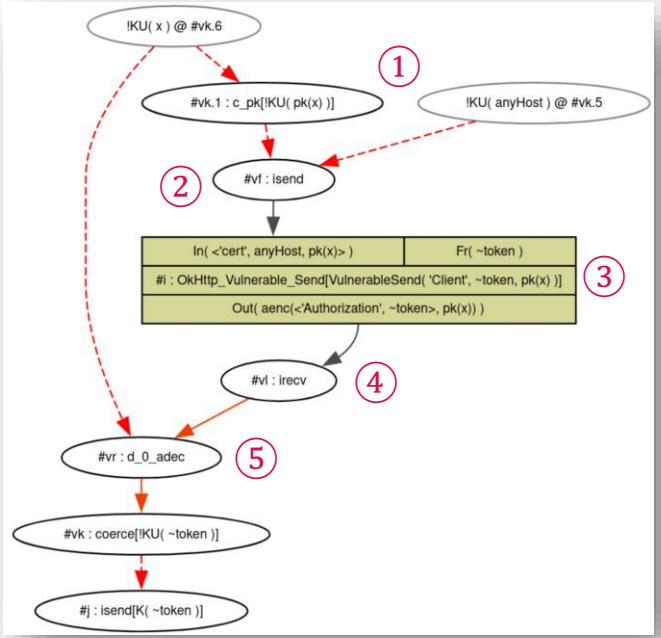
```
lemma Safe_Token_Is_Secret:
  "All token pk #i.
      SafeSend('Client', token, pk) @ #i
      ==>
      not(Ex #j. K(token) @ #j)
  "
```

От теории к практике – проверка леммы

```
lemma Vulnerable_Token_Is_Secret:
  all-traces
  "∀ token pk #i.
         (VulnerableSend( 'Client', token, pk ) @ #i) ⇒
         (¬(∃ #j. K( token ) @ #j))"
edit lemma or delete lemma
simplify
solve( !KU( ~token ) @ #vk.5 )
  case OkHttp_Vulnerable_Send
  solve( !KU( pk(x) ) @ #vk.5 )
    case c_pk
    SOLVED // trace found
  qed
qed
```

```
lemma Safe_Token_Is_Secret:
  all-traces
  "∀ token pk #i.
         (SafeSend( 'Client', token, pk ) @ #i) ⇒
         (¬(∃ #j. K( token ) @ #j))"
edit lemma or delete lemma
simplify
solve( !Pk( 'api.example.com', pk ) ▶₀ #i )
  case Register_Server
  solve( !KU( ~token ) @ #vk )
    case OkHttp_Safe_Send
    by solve( !KU( ~ltkS ) @ #vk.1 )
  qed
qed
```

От теории к практике – трасса атаки



- Злоумышленник знает свой ЗК и адрес своего сервера;
- Злоумышленник направляет параметры клиенту;
- Клиент шифрует токен аутентификации в сторону ОКШ злоумышленника;
- Злоумышленник получает и расшифровывает токен.

Итоги

- Аналогично рассмотренному примеру можно закодировать параметры и экспортируемые функции СКЗИ.
- Сформировать леммы, покрывающие ограничения на вызов функций.
- Проверить реализацию и сделать вывод о корректном использовании функций.
- Данный подход позволяет спроектировать и построить «супер-белый» интерфейс СКЗИ.

Спасибо
за внимание!

РусКрипто XXVIII НАУЧНО-ПРАКТИЧЕСКАЯ КОНФЕРЕНЦИЯ

Тырнов Филипп
Инженер-аналитик, КриптоПро
tyrnov@cryptopro.ru