Nice to meet you :)

### The SRLabs heroes behind this research

<table>
<thead>
<tr>
<th>Louis Merlin</th>
<th>Gabriel Arnautu</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Security researcher, focusing on blockchain security and threat analysis</td>
<td>- Security researcher, conducting blockchain security reviews</td>
</tr>
<tr>
<td>- Develops fuzz testing tools</td>
<td>- Focuses on tool development for audit automation</td>
</tr>
</tbody>
</table>

**Karsten Nohl**
- Studied cryptography, not crypto! =)
- Founder of SRLabs
- Research track record in mobile network security
- Former CISO at large telcos
- Lives in Bangkok

**Louis Merlin**
- Security researcher, focusing on blockchain security and threat analysis
- Develops fuzz testing tools

**Gabriel Arnautu**
- Security researcher, conducting blockchain security reviews
- Focuses on tool development for audit automation
Blockchain technology, love it or hate it, is continuously evolving; Researchers are hardly keeping up.

Research question – How do we proactively find bugs in large blockchain ecosystems?

This talk discusses five types of common blockchain bugs, and how to find them.
Bugs are explosive in crypto: Single-line integer overflow caused cryptocurrency to implode

### Background
- YAM launched in 2020 and quickly attracted >$500 million in assets
- The project founders warned about its immaturity and the lack of security auditing
- A bug caused the coin to lose control of its on-chain governance feature

### Vulnerable code snippet in YAM Finance rebase logic
```solidity
... totalSupply = initSupply.mul(yamsScalingFactor);
emit Rebase(epoch, prevYamsScalingFactor, yamsScalingFactor);
return totalSupply;
}
```

The rebase function aims to maintain token price stability. However, **due to an integer overflow** it incorrectly calculates the `totalSupply`, resulting in an excessive reserve of minted tokens.

### Impact
- Efforts to regain control of the YAM treasury failed
- YAM’s total market capitalization dropped from $65 million to **zero** in a few hours
- This event shows how a single vulnerability in a single line of code can compromise a whole project and the consumer funds behind it

---

Vulnerable code snippet in YAM Finance rebase logic

Security Research Labs
Criminals cash-out blockchain programming bugs in two ways

<table>
<thead>
<tr>
<th>Financial criminals extract value from blockchains in two ways</th>
<th>Move value on chain</th>
<th>Diminish overall value of chain</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Find bug in blockchain project</strong></td>
<td>Force transactions to move tokens to hacker’s account</td>
<td>Try to sell stolen tokens</td>
</tr>
<tr>
<td><strong>Short-sell tokens</strong> <em>(That is: Bet that the price will go down)</em></td>
<td>Use bug to degrade availability or performance</td>
<td>Trust in project dops; Value of blockchain drops</td>
</tr>
<tr>
<td><strong>Cash out short-sell = get paid because the price dropped</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Out of scope for this presentation: Other methods criminals use to defraud blockchains
- Abuse of a chain’s business logic (*economic attacks*)
- Hacking the underlying IT infrastructure or web/mobile apps
- Social engineering and other attacks on blockchain endusers
- Financial scams
Agenda

- Intro to third-generation blockchains
  - Five types of blockchain hacking
  - Fuzzing blockchains effectively
Third-generation blockchains set out to solve scalability and interoperability

1st generation blockchains
- Technology to transact with one another (at a peer-to-peer level)
- Users do not need to rely on centralized entities such as banks
- The design only allows you to send, receive and trade assets

2nd generation blockchains
- Introduces smart contracts, self-executing agreements made between parties
- Allows executing agreements without relying on an expensive intermediary to manage it
- Behaves as digital ecosystem that other crypto projects operate on
- Does not scale well, slowing down transactions

3rd generation blockchains
- Solves 2nd gen issue of scalability by creating more parallel transaction and more storage
- Introduces interoperability, allowing blockchains to interact with one another
- Adds more flexibility towards networking, node, and runtime configuration, empowering custom-purpose blockchains

Our focus today:

Distributed Ledger (Bitcoin)
2008

Smart Contracts (Ethereum)
2013

Interoperability (e.g. Polkadot)
2017
Substrate is a framework to program “third-gen” chains

<table>
<thead>
<tr>
<th>Engineering challenge</th>
<th>Their solution</th>
</tr>
</thead>
</table>
| Developers found they were recreating much of the same functionality but with different limitations around **scale**, **governance**, **forks**, **interoperability**, and **upgrades** | Substrate in a nutshell provides  
- Tooling for development, deployment, debugging  
- Blockchains to upgrade forkless  
- Hot-swap components (pallets) such as the network stack, consensus, finality engine |

Substrate is actively used by **153 teams** building blockchain projects, making it a **relevant security research target**
Substrate is the foundation of different blockchain projects; Scalable methods and toolchain needed for vulnerability testing

Architecture of Substrate client

- P2P networking
- Native Runtime
- Wasm Runtime (our focus today)
- Consensus
- Storage
- Telemetry
- RPC

FRAME pallets

- Aura
- BABE
- GRANDPA
- Elections
- Utility
- Atomic Swap
- Sudo
- Multisig
- Identity
- Assets
- Contracts
- EVM
- Collective
- Treasury
- Elections Phragmen
- Democracy
- Randomness
- Timestamp
- Staking
- more ..

Runtime

- Aura
- GRANDPA
- Sudo
- Assets
- Collective
- Treasury
- Elections Phragmen
- Timestamp

Security Research Labs
Agenda

- Intro to third-generation blockchains

  **Five types of blockchain hacking**

- Fuzzing blockchains effectively
### Five types of hacking attacks are commonly possible against third-gen blockchains

<table>
<thead>
<tr>
<th>Example hacking goal</th>
<th>Bug type</th>
<th>Bug impact</th>
<th>Availability</th>
<th>Integrity</th>
<th>Attack scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaust resources</td>
<td>Wrongly-priced transactions</td>
<td>Spam a wrongly-priced transaction can cause a DoS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manipulate program flow</td>
<td>Unsafe arithmetic</td>
<td>Abuse an operation to edit values to your advantage</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Logic bugs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DoS nodes</td>
<td>Reachable panics</td>
<td>Cause every available node to panic and cause a DoS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abuse business logic</td>
<td>Incorrect usage of standard patterns</td>
<td>Abuse a misconfiguration to gain a financial advantage or cause DoS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slow down chain</td>
<td>Storage bloating</td>
<td>Reduce chain useability by filling its storage</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These hacks may be leveraged against a wide range of blockchain projects, as they do not require any secret information from the victim; and most configurations and program source code is open.
### Underpriced function calls enable resource exhaustion

<table>
<thead>
<tr>
<th>Hacking goal</th>
<th>Bug type</th>
<th>Bug impact</th>
<th>Avail.</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exhaust resources</td>
<td>Wrongly-priced transactions</td>
<td>Spam a wrongly-priced transaction can cause a DoS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Example:

**Hacker short-sells tokens and drives down chain value**

1. The cost ("weight") of some blockchain function was estimated too low
2. Hacker drives chain into loop by calling underpriced function repeatedly
3. The blockchain is frozen, loses credibility and value
Miscalculation of block execution time can cause network DoS

Scenario 1: Exhaust resources

Background info.
Resources available to blockchains are limited. These resources include memory usage, storage I/O, computation, transaction/block size and state database size.

Attack.
The hacker sends a malicious transaction to cause block execution taking too long.

*One unit of weight is one picosecond of execution time, that is $10^{-12}$ weight = 1 second, or 1,000 weight = 1 nanosecond, on fixed reference hardware.

The hacker abuses a vulnerability in the weight* calculation, causing the nodes to miss their chance to generate new blocks, resulting in network DoS.

A transaction is a piece of information that comes from outside the chain and is included in a block.
Hackers can craft and gossip a nested transaction causing nodes to miss their slots and fail at block production, and potentially halting the blockchain.

### Scenario 1: Vulnerable Code in `sudo_as` transaction

```rust
#[weight = (
    call.get_dispatch_info().weight
    .saturating_add(10_000)
    // AccountData for inner call origin accountdata.
    .saturating_add(<T::DbWeight::get().reads_writes(1, 1)),
    call.get_dispatch_info().class
)
fn sudo_as(origin,
    who: <T::Lookup as StaticLookup>::Source,

To receive the `sudo_as` transaction call dispatch class and weight, the getter function to receive the dispatch information that holds both weight and class is called twice.

Example: nesting `sudo_as` 41 times results in a call tree with $2^{40} = 1,099,511,627,776$ leaves.

Hacker nests transaction `sudo_as(sudo_as(sudo_as(...))))` causing exponential complexity of the get dispatch info weight calculation function.
<table>
<thead>
<tr>
<th>Hacking goal</th>
<th>Bug type</th>
<th>Bug impact</th>
<th>Avail.</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manipulate program flow</td>
<td>Unsafe arithmetic</td>
<td>Abuse an operation to edit values to your advantage</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Logic bug</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example:** Competitor sabotages credibility of the chain

1. The cost (“weight”) of some blockchain function is calculated based on the input’s length; an arithmetic overflow will trigger a “wrap around” behavior.
2. Competitor calls function with large input parameter, causing a huge discrepancy between the calculated weight and the computational work required.
3. The blockchain acts contrary to its intended programming, loses credibility and value.
Integer overflows can lead to financial loss or denial of service

Scenario 1: Vulnerable Code

```rust
// verify that relayer is paying actual dispatch weight
let actual_dispatch_weight: Weight = messages
 .values()
 .map(|lane_messages| lane_messages
 .messages
 .iter()
 .map(T::MessageDispatch::dispatch_weight)
 .sum::<Weight>()
 )
 .sum();

if allowed_dispatch_weight < actual_dispatch_weight {
    return Err(Error::<T, I>::InvalidMessagesDispatchWeight.into());
}
```

The hacker sends a malicious transaction with a large proof in one of these queues, causing a denial of service because of the “wrap around” behavior when `sum()` is executed.

Attack sequence

1. Hacker transmits a `receive_messages_proof` transaction containing a large `proof` struct and a high value for `messages_count`
2. Node that executes the block will timeout, missing their production slot

The hacker sends a malicious transaction with a large proof in one of these queues, causing a denial of service because of the “wrap around” behavior when `sum()` is executed.
Bonus vulnerability: Arithmetic overflow prevention code leads to logic bug

Scenario 1: Vulnerable Code

```rust
let messages_in_the_proof = end.checked_sub(begin)
    .and_then(|diff| diff.checked_add(1))
    .unwrap_or(0);

if messages_in_the_proof != messages_count {
    return Err(MessageProofError::MessagesCountMismatch);
}

let mut messages = Vec::with_capacity(end);
```

To circumvent overflows, developers will use “safe mathematic operations”, such as `checked_add` or `saturating_add`. These add a new layer of complexity: the code must handle edge cases properly.

Attack sequence

1. Hacker transmits a `receive_message_proof` transaction containing an `end` value of `u64::MAX` and `begin` and `message_count` of 0, causing a vector to be allocated that has size `u64::MAX`

2. Node that executes the block panics on trying to allocate the vector
Logic bugs can enable hackers to gain unfair advantages and rewards

<table>
<thead>
<tr>
<th>Hacking goal</th>
<th>Bug type</th>
<th>Bug impact</th>
<th>Avail.</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manipulate program flow</td>
<td>Logic bug</td>
<td>Abuse the logic of the chain against itself, gaining some reward in the process</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unsafe arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Hacker exploits logic bug present in source code to spoof their identity

1. Hacker claims a legitimate identity on the blockchain and monitors the network to figure out when the registrar will hand out its sentence.
2. Hacker requests a spoofed identity immediately between the time when the registrar submits their judgment and when it is accepted on-chain.
3. The hacker now controls a validated spoofed identity because the code did not check which identity was being validated.
Hackers can spoof their identity by re-setting it right before the judgement is given.

By monitoring the network, a hacker can figure out when a registrar will provide judgement on their identity and rush to include a transaction modifying their requested identity, with a higher tip so it gets executed before the judgement.

**Attack sequence**

1. **Hacker requests for legitimate identity** to be judged “hello, I am legit_identity and here is my proof”

2. **Registrar provides judgement on identity** “hello User, your identity is correct”

3. **Hacker requests for spoofed identity** “hello, I am spoofed_identity” with high tip, thereby running ahead of transaction (2)
Unhandled return values can cause the nodes to panic, allowing a hacker to DoS the chain.

<table>
<thead>
<tr>
<th>Hacking goal</th>
<th>Bug type</th>
<th>Bug impact</th>
<th>Avail.</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoS nodes</td>
<td>Reachable panics</td>
<td>Cause every available node to panic and cause a DoS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Hacker launches DoS attack against chain for supply-chain attack

1. Non-explicit handling of a function’s result assumes that it cannot return a “None” value
2. Hacker causes all the nodes to panic by calling a transaction with a high value as parameter
3. The blockchain is frozen, functionality is halted for projects using the chain, and the ecosystem loses credibility and value
Triggering Rust panic conditions can compromise chain availability

Scenario 1: **Vulnerable code**

```rust
fn get_dot_to_token(dot_amount: u128) -> u128 {
    dot_amount.checked_mul(T::DOTToTokenRate::get()).unwrap();
}
```

Rust chooses to panic when `None` is returned in order to avoid any unexpected behavior.
In such cases, the runtime assumes that it is **better to stop the program instead of using an unexpected value.**

**Attack sequence**

1. Hacker transmits a `create_bid` transaction containing an unusually high `dot_amount` (close to `u128::MAX`)
2. Node that executes the block crashes

Hacker transmits a `create_bid(CENTS, 5734123568823053662, 31768)` transaction containing an unusually high `dot_amount` (close to `u128::MAX`). The node that executes the block crashes trying to execute the block.

Security Research Labs
Wrong configuration in runtime allows hacker to fill up blockchain’s storage

<table>
<thead>
<tr>
<th>Hacking goal</th>
<th>Bug type</th>
<th>Bug impact</th>
<th>Avail.</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abuse business logic</td>
<td>Incorrect usage of standard patterns</td>
<td>Abuse a misconfiguration allows a hacker to gain a financial advantage or cause DoS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: **Hacktivist drives down credibility of chain**

1. Wrongly configured runtime parameter allows the creation of an account for a derisory amount of money
2. Hacktivist creates a large number of accounts for a small amount of money, cluttering the storage of the chain
3. The blockchain’s storage size increases, causing longer transaction times; it loses credibility and value
A bad runtime configuration can open vulnerabilities in the blockchain

Scenario 1: Vulnerable Code

```rust
pub const UNITS: Balance = 1_000_000_000_000;
pub const CENTS: Balance = UNITS / 100;

parameter_types! {
    pub const ExistentialDeposit: Balance = 0;
    pub const MaxLocks: u32 = 50;
    pub const MaxReserves: u32 = 50;
}
```

Each account is represented by an Account structure which keeps track of user's balance and Substrate specific reference counters, but it can also be enhanced with project specific parameters. This data structure lives in the storage of the blockchain if the account has a balance of at least ExistentialDeposit.

Setting an existential deposit of 1 means setting an existential deposit of 0.000000000001 UNITS, which is not enough to prevent spamming the creating of new accounts.

If ExistentialDeposit is set to a small value, a hacker could create a lot of accounts which will fill up the storage of the blockchain, using only a small amount of money for transaction costs.

Attack

- full account – balance(0)
- full account – balance(0)
- full account – balance(0)
- full account – balance(0)
- ...
## Non existing storage deposits allows hacker to fill up blockchain’s storage

<table>
<thead>
<tr>
<th>Hacking goal</th>
<th>Bug type</th>
<th>Bug impact</th>
<th>Avail.</th>
<th>Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow down chain</td>
<td>Storage bloating</td>
<td>Reduce chain useability by filling its storage</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example:

**Disgruntled insider drives down credibility of chain**

1. **Missing storage deposit for the creation of large database items allows spamming this process**
2. **Disgruntled insider creates many storage items for a small amount of money, cluttering chain storage**
3. **Chain storage increases, causing longer transaction times and harder operability; it loses credibility and value**
Insufficient storage deposits can allow a hacker to cheaply fill the blockchain storage with excessive accumulation of data within a blockchain network, leading to increased storage requirements and potential operational inefficiencies.

### Scenario 1: Vulnerable Code

```rust
pub storage RequiredStakeForStakeAndSlash: Balance = 1_000_000;
...
pub fn register(origin: OriginFor<T>, valid_till: T::BlockNumber) -> DispatchResult {
    let relayer = ensure_signed(origin)?;
    ...
}

RegisteredRelayers::<T>::try_mutate(&relayer, |maybe_registration| -> DispatchResult {
    let mut registration = maybe_registration
        .unwrap_or_else(|| Registration { valid_till, stake: Zero::zero() });
    ...
}
```

**Attack**

Spamming millions of `Bridges::register()` calls could result in **1GB of storage** filled for only ~USD 25’000 (compared to tens of millions of $ in other blockchains).
Agenda

- Intro to third-generation blockchains
- Five types of blockchain hacking

Fuzzing blockchains effectively
### Three analysis techniques to find blockchain bugs

<table>
<thead>
<tr>
<th></th>
<th>Static analysis</th>
<th>Fuzzing</th>
<th>Manual review</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>B</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>C</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>D</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>E</td>
<td>Working on it</td>
<td></td>
<td>✅</td>
</tr>
</tbody>
</table>

#### Take aways

- **Static analysis** should be done as part of development process, using tools such as `semgrep` and `dylint`.
- Security testing typically starts with fuzz testing, which is particularly strong in finding availability bugs.
- Before an economic launch, every project should also go through security auditing including manual review.
We created software to find bugs in all these categories, this is our fuzz engine.

1. **AFL++ and honggfuzz based coverage guided fuzz engine**
   - Mutate binary input and transform into transactions (=API calls)

2. **Powerful mutator**
   - Fuzzer applies random permutation to given binary string

3. **Substrate-based target**
   - Run extracted transactions/API calls against the target program on a predefined state

**Custom parser-based fuzzing:**
Transform binary stream of data into list of runtime calls and run against target program

**Seed pool**
- Mutate binary input and transform into transactions (=API calls)

**Seed**
- Selection

**Input**
- Mutate

**Target**
- Mutate binary input and transform into transactions (=API calls)

**Transform into transaction using encoder**
- SPLIT BY DELIMITER
  - aefd05
  - ********
  - bdca3b
  - ********
  - cdea12

**Run against target**
- Run extracted transactions/API calls against the target program on a predefined state

- **ADD TO SEED POOL IF ADDITIONAL COVERAGE**
- **DROP IF NO ADDITIONAL COVERAGE**

- **ADD TO SEED POOL IF ADDITIONAL COVERAGE**
- **DROP IF NO ADDITIONAL COVERAGE**

- **Transformer 3 DOTS From Alice to Bob**
- **Vote Yay on Proposal #5**
- **Nominate validator M with 2 Dots**
impl pallet_template::Config for Runtime {
    type RuntimeEvent = RuntimeEvent;
    type WeightInfo = pallet_template::weights::SubstrateWeight<Runtime>;
}

// Create the runtime by composing the FRAME pallets that were previously configured.
construct_runtime!
{
    pub struct Runtime
    where
        Block = Block,
        NodeBlock = opaque::Block,
        UncheckedExtrinsic = UncheckedExtrinsic,
    {
        System: frame_system,
        Timestamp: pallet_timestamp,
        Aura: pallet_aura,
        Grandpa: pallet_grandpa,
        Balances: pallet_balances,
        TransactionPayment: pallet_transaction_payment,
        Sudo: pallet_sudo,
        // Include the custom logic from the pallet-template in the runtime.
        TemplateModule: pallet_template,
    }
}

/// The address format for describing accounts.
pub type Address = sp_runtime::MultiAddress<AccountId, ()>;

/// Block header type as expected by this runtime.
pub type Header = generic::Header<BlockNumber, BlakeTwo256>;

/// Block type as expected by this runtime.
pub type Block = generic::Block<'_, Header, UncheckedExtrinsic>;

/// The SignedExtension to the basic transaction logic.
pub type SignedExtra = (frame_system::CheckNonZeroSender<Runtime>,

https://github.com/srlabs/substrate-runtime-fuzzer

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
<th>Commits</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>kitchensink-fuzzer</td>
<td>Update substrate to polkadot-v0.9.43</td>
<td>d3d67d9</td>
<td>last month</td>
</tr>
<tr>
<td>kusama-fuzzer</td>
<td>Update substrate to polkadot-v0.9.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node-template-fuzzer</td>
<td>Update substrate to polkadot-v0.9.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>statemine-fuzzer</td>
<td>Update substrate to polkadot-v0.9.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.gitignore</td>
<td>First push</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cargo.toml</td>
<td>Add release build of certain dependencies</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LICENSE-APACHE</td>
<td>First push</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LICENSE-MIT</td>
<td>First push</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
https://github.com/srlabs/ziggy
We continuously find bugs on a variety of chains

<table>
<thead>
<tr>
<th>Issue type</th>
<th>Number of security issues found in 45 security reviews since January 2022</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrongly-priced transactions</td>
<td>23</td>
</tr>
<tr>
<td>Unsafe arithmetic</td>
<td>18</td>
</tr>
<tr>
<td>Reachable panic issues</td>
<td>15</td>
</tr>
<tr>
<td>Memory issues</td>
<td>7</td>
</tr>
<tr>
<td>Configuration issues</td>
<td>12</td>
</tr>
<tr>
<td>Cryptography issues</td>
<td>9</td>
</tr>
<tr>
<td>Improper authentication</td>
<td>2</td>
</tr>
<tr>
<td>Storage issues</td>
<td>12</td>
</tr>
</tbody>
</table>

Semi-automated testing is most effective in detecting insufficient benchmarking, unsafe arithmetic usage, reachable panics and configuration issues.
Takeaways

1. Blockchains contain fascinating hacking puzzles
2. Most bugs fall into five categories, many are crashes
3. Open-source tools enable mostly-automated reviews

Questions?

Karsten Nohl <nohl@srlabs.de>
Louis Merlin <louis@srlabs.de>
Gabriel Arnautu <gabriel@srlabs.de>