

Cracking Java's Armor: Fuzzing the JVM for Security Weaknesses

Sergey Temnikov
vulnerability researcher

```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Create the main window  
root = tk.Tk()  
root.title("Simple Keyboard")
```

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```

01

Intro

```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Run the application  
root.mainloop()
```

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



About me

Over **20** years of experience in IT security.

- Previously worked at **SEC Consult**, **Kaspersky** and **AWS**
- **Bug bounty** hunter with a strong focus on vulnerability research

Areas of Interest

- Fuzzing
- Java
- PHP
- Web3

Why Java Security Matters?

- Java is one of the most widely used programming languages in the world, and many vendors across various industries rely on it for building applications, platforms, and systems (**Oracle,Google,Amazon**)
- High-profile incidents like the **Log4Shell** vulnerability (**CVE-2021-44228**) have demonstrated how **JVM**-related exploits can disrupt global systems, highlighting the critical need for robust security measures.
- Security-critical applications (banking, government, IoT and Mobile Phones)

Java and the Java Virtual Machine (JVM)

- Java is a platform-independent programming language.
- Java code is compiled into **bytecode**, which is not directly executed by the operating system.
- The Java Virtual Machine (**JVM**) interprets and executes the bytecode
- **"Write Once, Run Anywhere"** philosophy has made Java a cornerstone of modern software development
- **JVM** provides features like automatic memory management (Garbage Collection) and runtime optimizations (**JIT** Compilation)

Understanding Java in This Context

When we talk about Java in this presentation, we are not just referring to the Java programming language itself. Instead, we mean the broader **JVM** ecosystem, which includes:

- Java
- Scala
- Kotlin
- Groovy
- Clojure
- JRuby

Java and native code

Java is designed to be platform-independent, but for performance-critical tasks, it integrates native code written in C or C++. This helps optimize execution speed by leveraging system-level optimizations.

- **JPEG & Image Processing:** Java uses native libraries like libjpeg for efficient image decoding, resizing, and manipulation.
- **Cryptography:** Java's security libraries (e.g., OpenJDK's **sun.security.provider.NativePRNG**) use native implementations for encryption and random number generation.
- **Graphics & UI:** JavaFX and AWT use native rendering pipelines for better performance.
- **I/O Operations:** The Java NIO (New Input/Output) framework uses direct memory access and system calls via native code for fast file and network handling.

Java and native code

There can be different implementations of common JVM built-in components.

```
C:\Users\spacewasp\Downloads\jdk-master\src\java.base  
n      Name  
..  
aix  
linux  
macosx  
share  
unix  
windows
```

JVM implementations

Most vendors have their own **JVM** implementations.

- Oracle HotSpot JVM
- OpenJDK JVM
- Amazon Corretto
- Azul Zulu
- GraalVM
- IBM J9
- SAP JVM
- Red Hat OpenJDK

Overview of JVM Attack Surface

- **Bytecode** execution
- Just-In-Time (**JIT**) compilation
- Garbage Collection (**GC**)
- Native interface (**JNI**)
- Class loading & sandboxing

Java application as an attack surface

- Attackers may exploit vulnerabilities within the **application logic**, business logic flaws, or insecure **third-party** libraries used within the app.
- Common entry points include:
 - input validation
 - deserialization
 - improper access control
 - unsafe interactions with external systems.

```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Run the application  
root.mainloop()
```

Basics of Fuzzing

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



An Introduction to Fuzzing

Automated input generation to detect crashes, memory corruption, or security vulnerabilities.



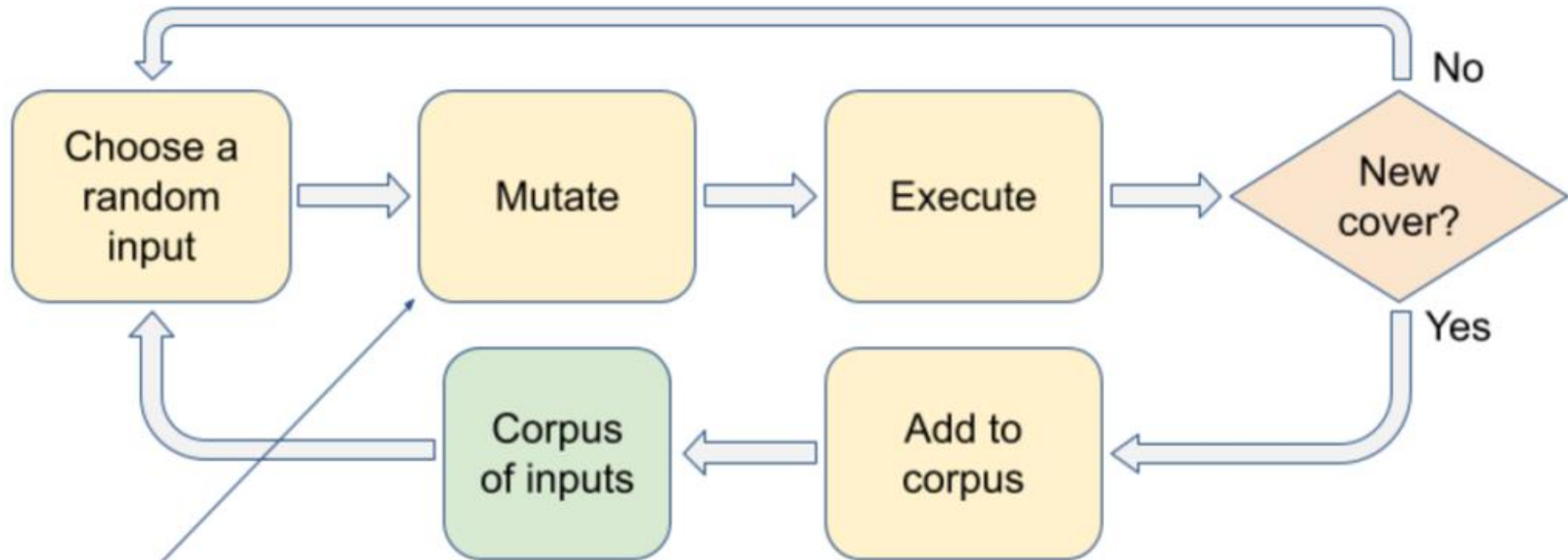
Simple steps

- 1) ``cat /dev/random > ./pdf_reader``
- 2) `crash???` $\rightarrow$ `save crash`
- 3) `goto 1`

Types of Fuzzing

- Dumb **vs** Smart
- Coverage-guided fuzzing (e.g., **libFuzzer**, **AFL**)
- Symbolic execution and hybrid fuzzing

Coverage-guided fuzzing



Mutate according to the structure (e.g. insert/remove XML tags)

Symbolic execution

```
1  int f() {  
2      ...  
3      y = read();  
4      z = y * 2;  
5      if (z == 12) {  
6          fail();  
7      } else {  
8          printf("OK");  
9      }  
10 }
```

Artifacts of Fuzzing

- Native crashes
- Memory leakage
- Infinite loops

Coverage-Guided Fuzzing

Enhancing Test Effectiveness through Code Coverage via Code Instrumentation



Static Instrumentation

```
int LLVMFuzzerTestOneInput(const uint8_t
*data, size_t size)
{
    if ( size )
    {
        if ( data[0] == 'H' )
        {
            if ( size > 1 )
            {
            }
        }
    }
    return 0;
}
```

```
int LLVMFuzzerTestOneInput(const uint8_t *data,
size_t size)
{
    ++__start__sancov_cntrs;
    _sanitizer_cov_trace_const_cmp8(0, size);
    if ( size )
    {
        _sanitizer_cov_trace_const_cmp4('H',
data[0]);
        if ( data[0] == 'H' )
        {
            _sanitizer_cov_trace_const_cmp8(1, size);
            if ( size > 1 )
            {
            }
            else
            {
                ++counter_171E8B;
            }
        }
        else
        {
            ++counter_171E8A;
        }
    }
    else
    {
        ++counter_171E89;
    }
    return 0;
}
```

Static Instrumentation

- **LibFuzzer** (llvm clang)
- **AFL** clang/gcc

Dynamic Instrumentation

Black-box Fuzzing: Testing without knowledge of internal code. Runtime instrumentation

- **DynamoRIO**
- **WinAFL**
- **QEMU**
- Intel Instrumentation and Tracing Technology

Fuzzing: What Should (Not) Be Fuzzed

- **PREFERRED:** Standalone libraries (easier to isolate and test)
- **YES:** Parsers and loaders (high-risk attack surface)
- **NO:** Targets with outgoing network activity outside the fuzzing environment
- **REQUIRED:** Deterministic behavior (no random data that prevents reproducibility)
- **WITH CAUTION:** APIs that can block processes (e.g., wait/select/read...)
- **WITH CAUTION:** Cache mechanisms (may affect fuzzing efficiency and results)

```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Run the application  
root.mainloop()
```

Fuzzing the JVM: Challenges & Approaches

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



JVM bytecode

```
// Bytecode stream: 03 3b 84 00 01 1a 05
68 3b a7 ff f9
// Disassembly:

iconst_0      // 03
istore_0      // 3b
iinc 0, 1     // 84 00 01
iload_0       // 1a
iconst_2      // 05
imul          // 68
istore_0      // 3b
goto -7       // a7 ff f9
```

Instrumenting VM Code Instead of the VM

- Fuzzers need feedback from **executed code** to generate better test cases.
- Instrumenting **bytecode** execution provides coverage-guided feedback.
- Improves fuzzing efficiency by focusing on executed Java code paths rather than entire **JVM**.

Unique Challenges in Fuzzing JVM

- Memory safety issues **vs** logic bugs
- Garbage collection **affecting** fuzzing results
- Memory consumption by **static variables**
- **Race** conditions

Challenge

```
1 int f() {  
2     ...  
3     if( read() == md5(builtin_password) ) {  
4         printf("You are admin");  
5     }  
6 }
```

Approaches to JVM Fuzzing

- Fuzzing **Bytecode** Interpreters
- Deserialization/Parser/Readers
- Java Native Interface (**JNI**) Fuzzing
- Generating and Executing Random Java Code to Identify Bugs

Existing Tools for JVM Fuzzing

- **Jazzer** (Java Fuzzer)
- **JQF**
- AFL++ with Java Bytecode
- SymQEMU + JVM
- Custom harnesses with libFuzzer

Jazzer

<https://github.com/CodeIntelligenceTesting/jazzer>

<https://www.code-intelligence.com/>



Jazzer. Results

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 1434179311
INFO: Loaded 1 modules (8 inline 8-bit counters): 8 [0x5f03d189be90, 0x5f03d189be98),
INFO: Loaded 1 PC tables (8 PCs): 8 [0x5f03d189be98, 0x5f03d189bf18),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 2 ft: 2 corp: 1/1b exec/s: 0 rss: 31Mb
#144    NEW    cov: 3 ft: 3 corp: 2/2b lim: 4 exec/s: 0 rss: 31Mb L: 1/1 MS: 2 ChangeByte-ChangeByt
#157    NEW    cov: 4 ft: 4 corp: 3/4b lim: 4 exec/s: 0 rss: 31Mb L: 2/2 MS: 3
CrossOver-ChangeBit-CrossOver-
#1345   NEW    cov: 5 ft: 5 corp: 4/8b lim: 14 exec/s: 0 rss: 32Mb L: 4/4 MS: 3
InsertByte-ChangeBit-CrossOver-
#1696   NEW    cov: 6 ft: 6 corp: 5/10b lim: 17 exec/s: 0 rss: 32Mb L: 2/4 MS: 1 EraseBytes-
#1832   REDUCE cov: 6 ft: 6 corp: 5/9b lim: 17 exec/s: 0 rss: 32Mb L: 3/3 MS: 1 EraseBytes-
```

```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Run the application  
root.mainloop()
```

Case Studies of JVM Vulnerabilities

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



Vulnerability Types

- **Binary:** Memory corruption, native code execution...
- **Logical:** type confusion, deserialization flaws, SQL injection...

Vulnerabilities: Binary

```
#include <stdlib.h>

int main(){
    ... int32_t* m = (int32_t*)malloc(0x8000000000000000LL);
    ... m[0] = 1;
    ... free(m);
    ... return 0;
}
```

Vulnerabilities: Logical

```
public boolean login(String username, String password). {}  
    String sql =  
        "select * from users where username = " + "'" + username + "'" + " and " +  
        "password " + " = " + "'" + password + "'";  
  
    ResultSet result = stmt.executeQuery(query);  
  
    if(result.next()) {  
        /* Login Success */  
        return true;  
    }  
    else. {  
        /* Login Failed */  
        return false;  
    }  
}
```

Java AWT

Multiple naive heap corruption vulnerabilities in **AWT** (via **Apache Batik**)

~**15** different native **crashes** and **DOS**

Java AWT. SVG

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/
<svg enable-background="new 0 0 2014 1674.641" version="1.1" viewBox="0 0 2014 1674.641" :
    <path d="m44.883 1173.3c0 169.58 92.136 317.37 229.08 396.59v-793.53c-136.94 79.2
    <path d="m1496.5 173.21c-338.54-181.18-640.46-181.19-978.98 0-115.34 61.726-232.1
171.6 225.54 272.38v1048h125.99v-1048c0-147.36-116.76-286.94-232.1-348.66z"/>
    <path d="m1740 776.38v793.53c136.94-79.214 229.08-227 229.08-396.59 1e-3 -169.57-
    <path d="m541.72 709.03h-70.645c-27.201 0-49.252 22.051-49.252 49.253v829.79c0 27
    <path d="m1542 709.03h-70.646c-27.2 0-49.252 22.051-49.252 49.253v829.79c0 27.201
    <path d="m595.97 1132.8c4.322-0.219 8.643-0.184 12.965 0 0.197 4.557 3.624 24.404
38.746h10.467c10.735-37.493 8.785-86.788 9.941-115.01 1.07-3.103 0.614-6.426 0.868-9.634
1.376-9e-3 0.421-1.771 0.877-3.515 1.315-5.277l0.657-9e-3c0.202 4.611-0.719 9.38 0.684 13
```

Java AWT. Code generation

```
public void paintComponent(Graphics g) {  
    Graphics2D g2d = (Graphics2D) g;  
    int x1 = random.nextInt(getWidth());  
    int y1 = random.nextInt(getHeight());  
    int x2 = random.nextInt(getWidth());  
    int y2 = random.nextInt(getHeight());  
    int size = random.nextInt(50) + 10;  
    g.setColor(new Color(random.nextInt(256), random.nextInt(256), random.nextInt(256)));  
    g.fillOval(x1, y1, size, size);  
    g.setColor(new Color(random.nextInt(256), random.nextInt(256), random.nextInt(256)));  
    g.drawOval(x2, y2, size, size);  
    g.setColor(new Color(random.nextInt(256), random.nextInt(256), random.nextInt(256)));  
    g2d.setStroke(new BasicStroke(3));  
    g.drawLine(x1, y1, x2, y2);  
    g.setColor(new Color(random.nextInt(256), random.nextInt(256), random.nextInt(256)));  
    g.fillRect(random.nextInt(getWidth()), random.nextInt(getHeight()), size, size);  
    g.setColor(new Color(random.nextInt(256), random.nextInt(256), random.nextInt(256)));  
    g.drawRect(random.nextInt(getWidth()), random.nextInt(getHeight()), size, size);  
}
```

Freetype (heap-buffer-overflow)

2022-09-16 Dominik Röttsches <drott@chromium.org>

Reject 'COLR' table if version is 1 but header too small.

* src/sfnt/ttcolr.c (tt_face_load_colr): If the version is determined to be 1, then the table size has to be at least the size of the v1 header. Also, for peeking the number of base glyphs and entries in the layer list, ensure that the table is sufficiently long.

Fixes #1179. Original patch by Sergey Temnikov.

Apache Xerces Java Parser

Posted to j-dev@xerces.apache.org



Mukul Gandhi - Monday, January 24, 2022 2:16:23 PM GMT+1

To: announce@apache.org, j-dev@xerces.apache.org

Reply-To: j-dev@xerces.apache.org

Subject: CVE-2022-23437: Infinite loop within Apache XercesJ xml parser

Severity: high

Description:

There's a vulnerability within the Apache Xerces Java (XercesJ) XML parser when handling specially crafted XML document payloads. This causes, the XercesJ XML parser to wait in an infinite loop, which may sometimes consume system resources for prolonged duration. This vulnerability is present within XercesJ version 2.12.1 and the previous versions.

Mitigation:

Apache XercesJ users, should migrate to version 2.12.2

Credit:

This issue was discovered by Sergey Temnikov and Ziyi Luo, from Amazon Corretto/JDK Team

Apache Xerces Java Parser

```
public class XMLParserSAX {  
    public static void main(String[] args) throws Exception {  
        String xmlData = "<root><item id='1'><name>Example</name></item></root>";  
        ByteArrayInputStream inputStream = new ByteArrayInputStream(xmlData.getBytes());  
  
        SAXParserFactory factory = SAXParserFactory.newInstance();  
        SAXParser saxParser = factory.newSAXParser();  
        saxParser.parse(inputStream, new MyDummyHandler());  
    }  
}
```

Apache Parquet Parser

CVE-2021-41561: Apache Parquet-MR potential DoS in case of malicious Parquet file

Posted to dev@parquet.apache.org



Gábor Szádovszky - Monday, December 20, 2021 11:03:38 AM GMT+1

Description:

Improper Input Validation vulnerability in Parquet-MR of Apache Parquet allows an attacker to DoS by malicious Parquet files. This issue affects Apache Parquet-MR version 1.9.0 and later versions.

This issue is being tracked as PARQUET-2094

Mitigation:

1.12.x users should upgrade to 1.12.2

1.11.x users should upgrade to 1.11.2

Users of older release lines ($\leq 1.10.x$) should upgrade to 1.12.2 or 1.11.2

Credit:

This issue was discovered by Sergey Temnikov of the Amazon S3 team.

05



```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Run the application  
root.mainloop()
```

Setting Up a JVM Fuzzing Environment

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



Jazzer Fuzzing Challenges

- Mutators
- Sanitizers
- Memory exhausting
- Corpus exhausting

Built-in Jazzer Sanitizers

1. Deserialization
2. ExpressionLanguageInjection
3. FilePathTraversal
4. LdapInjection
5. OsCommandInjection
6. ReflectiveCall
7. RegexInjection
8. ScriptEngineInjection
9. ServerSideRequestForgery
10. SqlInjection

Running and Analyzing Results

- Crashes and logs
- Reproducing and debugging
- Writing Proof-of-Concept (**PoC**) exploits



```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Run the application  
root.mainloop()
```

Future of JVM Security & Fuzzing

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



Limitations of Current Fuzzing Techniques

- Difficulties in achieving **deep coverage**
- Difficulties in finding the proper **entry point**

Next Steps in JVM Fuzzing

- **AI-powered** fuzzing techniques
- Hybrid fuzzing with symbolic execution

07



```
def key_pressed(key):  
    print(f"{key} pressed")
```

```
# Run the application  
root.mainloop()
```

Conclusion

```
# Create buttons for keys  
keys = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P']  
in the following keys:  
    button = tk.Button(root, text=key, width=5,  
        command=lambda k=key: key_pressed(k))  
    button.pack(side=tk.LEFT)
```



Key Takeaways

- **JVM** is a high-value target for attackers
- Fuzzing is effective for finding **hidden weaknesses**
- Combining fuzzing with other techniques **enhances security**
- Fuzzer is **NOT** a standalone solution; it's a **building block** for creating a custom one

Contacts

Mail: spacewasp1982@gmail.com

Linkedin: <https://me.linkedin.com/in/sergey-temnikov-65b47>



positive Cairo hack talks

pt

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Display the cube and voxel
count
fig, ax = plt.subplots
(subplot_kw={"projection": "3d"})
ax.voxels(cube, edgecolor='k')
ax.set_title(f"Cube with
{np.count_nonzero(voxels)}
voxels")
plt.show()
```

Thank you

```
# Create a 3D cube
cube = np.zeros((10, 10, 10), dtype=bool)
cube[2:8, 2:8, 2:8] = True
```

