Intermediate Level

Advanced SSRF Attack Techniques

Bypassing Common Defenses

Blacklist Bypass Techniques:

- **IP Address Obfuscation:**
  - **Decimal notation: http://2130706433/ (127.0.0.1)**
  - **Octal notation: http://0177.0.0.1/**
  - **Hexadecimal notation: http://0x7f000001/**
  - **IPv6 mapping: http://[::ffff:127.0.0.1]/**
  - **IPv6 compressed: http://[::1]/**
  - **IPv6 with embedded IPv4: http://[0:0:0:0:0:ffff:127.0.0.1]/**
  - **Dword notation: http://0x7f.0x0.0x0.0x1/**
  - **Zero suppression: http://127.1/**
  - **Octal with padding: http://0127.0.0.1/**
- **Domain Obfuscation:**
  - **Using subdomains: http://localhost.attacker.com/**
  - **Registering domains containing target names**
  - **Using encoded domains: http://xn--80a.com/ (Punycode)**
  - **Using domains that resolve to internal IPs**
- **URL Encoding Tricks:**
  - **Double encoding: http://127.0.0.1/ → http%3A%2F%2F127.0.0.1%2F**
  - **Mixed encoding: http://12%37.0.0.1/**
  - **UTF-8 characters: http://①②⑦.⓪.⓪.①/**
  - **Unicode normalization: Using visually similar characters**
- **Redirection Techniques:**
  - **Open redirects: https://example.com/redirect?url=http://internal-service/**
  - **301/302 redirects to internal URLs**
  - **Meta refresh redirects**
  - **JavaScript redirects**
  - **Short URLs that resolve to internal addresses**

Protocol Smuggling:

- **Scheme mixups: gopher://127.0.0.1:25/**
- **URL protocol handler abuse: java:Runtime.getRuntime().exec('command')**
- **Less common protocols:**
    - **dict:// (Dictionary Server Protocol)**
    - **sftp:// (SSH File Transfer Protocol)**
    - **ldap:// (Lightweight Directory Access Protocol)**
    - **tftp:// (Trivial File Transfer Protocol)**
    - **rtsp:// (Real Time Streaming Protocol)**

Cloud-Specific Bypass Techniques:

- **Alternative metadata service endpoints:**
    - **AWS: http://169.254.169.254/**
    - **GCP: http://metadata.google.internal/**
    - **Azure: http://169.254.169.254/metadata/instance**
    - **Digital Ocean: http://169.254.169.254/metadata/v1/**
- **Alternative metadata service paths:**
    - **AWS: /latest/meta-data/ vs /latest/user-data/**
    - **GCP: /computeMetadata/v1/ with Metadata-Flavor header**
    - **Undocumented metadata endpoints**

DNS Rebinding Techniques:

- **Fast-flux DNS: Rapidly changing IP resolution**
- **Time-based rebinding: Valid IP initially, internal IP later**
- **Multiple-Answer rebinding: Returns multiple IPs including internal ones**
- **Advanced tools like Singularity for automated DNS rebinding**
- **Rebinding with short TTLs to bypass DNS caching**

Advanced SSRF Exploitation

Internal Service Enumeration:

- **Port scanning via SSRF:**
    - **Timing-based port scanning**
    - **Error-based detection**
    - **Response size analysis**
    - **TCP vs UDP port scanning**

    - **Banner grabbing**
- **Service Fingerprinting:**
    - **Analyzing response patterns**
    - **Identifying web servers, application servers**
    - **Detecting common admin interfaces**
    - **Finding application version information**

Exploiting Specific Services:

- **Redis exploitation via SSRF:**
    - **Gopher protocol for raw TCP**
    - **Writing to authorized_keys for access**
    - **Configuration manipulation**
    - **Data extraction**
- **Memcached exploitation:**
    - **Data extraction**
    - **Cache poisoning**
    - **Revealing application secrets**
- **Elasticsearch exploitation:**
    - **Reading sensitive indices**
    - **Extracting configuration**
    - **Modifying data if writable**
- **Docker exploitation:**
    - **Accessing Docker API**
    - **Creating privileged containers**
    - **Executing commands in containers**

Complex SSRF Chains:

- **SSRF to XXE:**
    - **Accessing internal XML parsers**
    - **Triggering XXE via internal services**
    - **Reading local files through XXE**
- **SSRF to RCE:**
    - **Exploiting local command injection points**

    - **Accessing vulnerable admin interfaces**
    - **Exploiting unpatched internal services**
    - **Attacking deployment systems**
- **SSRF to CSRF:**
    - **Triggering CSRF on internal applications**
    - **Changing configurations**
    - **Executing privileged operations**

Out-of-Band Exploitation:

- **DNS for data exfiltration:**
    - **Encoding data in subdomain names**
    - **Using DNS resolution for blind SSRF confirmation**
    - **Creating DNS tunnels for command and control**
- **HTTP callbacks for confirmation:**
    - **Using unique identifiers in requests**
    - **Blind SSRF validation via external callbacks**
    - **Burp Collaborator or similar tools for monitoring**
- **Time-based techniques:**
    - **Measuring response times to infer results**
    - **Sleep-based confirmation**
    - **Resource exhaustion techniques**

Intermediate Defense Mechanisms

Improved Validation Approaches:

- **Context-aware URL validation:**
    - **Parse and validate each URL component**
    - **Validate against hostname patterns, not just exact matches**
    - **Domain suffix validation**
    - **Implementing URL parsing correctly across frameworks**
- **Defense-in-depth validation:**
    - **Combining multiple validation strategies**
    - **Multi-stage validation pipeline**
    - **Pre and post-parsing validation**

    - **Static and dynamic validation**

Network-Level Controls:

- **Microsegmentation:**
    - **Strict network boundaries between services**
    - **Zero-trust network model**
    - **Application-level firewalls**
    - **Host-based firewalls**
- **Service-specific firewall rules:**
    - **Restricting outbound connections**
    - **Protocol-level filtering**
    - **Blocking connections to internal services**
    - **Explicit allow-listing for external services**
- **Proxies and Gateways:**
    - **Dedicated forward proxies for external requests**
    - **Gateway services for third-party interactions**
    - **URL rewriting and sanitization**
    - **Response filtering**

Language and Framework-Specific Protections:

- **Java:**
    - **Using java.net.URL whitelist**
    - **URLConnection restrictions**
    - **Custom protocol handler limitations**
    - **Java Security Manager configurations**
- **Python:**
    - **urllib safe handling**
    - **Requests library with proxy settings**
    - **Custom adapter implementation**
    - **Protocol restriction**
- **Node.js:**
    - **HTTP client restrictions**
    - **URL parsing protections**

    - **Custom Agent implementation**
    - **Protocol whitelist enforcement**
- **Ruby:**
    - **Net::HTTP safeguards**
    - **Open-uri restrictions**
    - **Faraday middleware protections**
    - **Custom URL validation**
- **PHP:**
    - **stream_context_create options**
    - **curl_setopt restrictions**
    - **allow_url_include and allow_url_fopen settings**
    - **Custom wrapper restrictions**

Advanced Application Architecture:

- **Mediator services:**
    - **Dedicated services for external communication**
    - **Enforcing strict contracts for requests**
    - **Response sanitization and filtering**
    - **No direct access to internal network**
- **URL tokenization:**
    - **Pre-registering allowed destinations**
    - **Using signed tokens for approved URLs**
    - **Time-limited URL tokens**
    - **Encrypted URL parameters**
- **Request signing:**
    - **Cryptographic validation of request destinations**
    - **Chain of trust for request forwarding**
    - **HMAC-based URL validation**
    - **Signed request parameters**

Intermediate Detection and Response

Enhanced Detection Patterns:

- **Contextual detection:**

    - **Understanding normal application traffic patterns**
    - **Detecting anomalous request destinations**
    - **Profiling allowed external services**
    - **Baselining typical request patterns**
- **Behavioral analysis:**
    - **Monitoring request frequency patterns**
    - **Connection duration analysis**
    - **Data volume monitoring**
    - **Request path analysis**

WAF and Runtime Protection:

- **Custom WAF rules:**
    - **Detecting SSRF patterns in parameters**
    - **IP-based blocking for internal destinations**
    - **Protocol restriction enforcement**
    - **Behavioral rules for request patterns**
- **RASP solutions:**
    - **Runtime detection of SSRF attempts**
    - **Context-aware blocking**
    - **Real-time intervention**
    - **Integration with existing security tools**

Incident Response for SSRF:

- **Containment strategies:**
    - **Temporarily restricting external requests**
    - **IP blocking for suspected attack sources**
    - **Disabling vulnerable functionality**
    - **Routing suspicious traffic through inspection proxies**
- **Forensic analysis:**
    - **Identifying compromised endpoints**
    - **Tracing request paths through systems**
    - **Analyzing exposed data**
    - **Determining attack duration and scope**

Intermediate Implementation Examples

Improved Validation in Python

python

```
import re
import socket
import ipaddress
from urllib.parse import urlparse
from requests import get

def is_valid_external_url(url):
    """
    Comprehensive URL validation for SSRF prevention
    """
    try:
        # Basic URL format validation
        parsed = urlparse(url)
        if not all([parsed.scheme, parsed.netloc]):
            return False

        # Scheme validation - restrict to HTTP/HTTPS
        if parsed.scheme not in ['http', 'https']:
            return False

        # Extract hostname and check against private IP ranges
        hostname = parsed.netloc.split(':')[0]

        # Check for localhost in different formats
        localhost_patterns = [
            r'^localhost$',
            r'^127\.',
            r'^::1$',
```

```
        r'^0\.0\.0\.0$',
        r'^\.local$'
    ]

    for pattern in localhost_patterns:
        if re.search(pattern, hostname, re.IGNORECASE):
            return False

    # Resolve hostname to check for internal IPs
    try:
        ip_addresses = socket.getaddrinfo(hostname, None)
        for addr_info in ip_addresses:
            family, _, _, _, socket_addr = addr_info
            ip_str = socket_addr[0]

            # Convert to IPv4Address or IPv6Address object
            ip = ipaddress.ip_address(ip_str)

            # Check if IP is private or loopback
            if ip.is_private or ip.is_loopback or ip.is_link_local:
                return False

    except socket.gaierror:
        # Failed to resolve - could be invalid hostname
        return False

    # Validate against allowed domains (whitelist)
    allowed_domains = ['api.example.com', 'cdn.example.org']
    if hostname not in allowed_domains and not any(hostname.endswith('.' + domain) for domain
in allowed_domains):
        return False
```

```
        return True

    except Exception as e:
        # Any parsing or validation error should fail closed
        print(f"URL validation error: {e}")
        return False

def fetch_url_safely(url):
    """
    Safely fetch URL content with proper validation
    """
    if not is_valid_external_url(url):
        return "Invalid URL or internal resource requested"

    try:
        # Use a timeout to prevent long-running requests
        response = get(url, timeout=10,
                       # Don't follow redirects to prevent bypass
                       allow_redirects=False,
                       # Limit response size
                       stream=True)

        # Check redirect attempts
        if response.status_code in [301, 302, 303, 307, 308]:
            return "Redirects are not allowed"

        # Limit response size
        content = response.raw.read(10 * 1024 * 1024)  # 10MB limit

        return content
```

```
    except Exception as e:
        return f"Error fetching URL: {str(e)}"
```

Proxy-Based Architecture (Node.js)

javascript

```
const express = require('express');
const { URL } = require('url');
const axios = require('axios');
const ipRangeCheck = require('ip-range-check');
const dns = require('dns');
const { promisify } = require('util');

const app = express();
const dnsLookup = promisify(dns.lookup);

// Whitelist of allowed domains
const ALLOWED_DOMAINS = ['api.trusted-service.com', 'cdn.approved-domain.org'];

// Forbidden IP ranges
const FORBIDDEN_RANGES = [
  '10.0.0.0/8',
  '172.16.0.0/12',
  '192.168.0.0/16',
  '127.0.0.0/8',
  '0.0.0.0/8',
  '169.254.0.0/16',
  'fc00::/7',
  '::1/128'
];

async function validateUrl(urlString) {
  try {
```

```
// Parse the URL
const url = new URL(urlString);

// Check scheme
if (url.protocol !== 'http:' && url.protocol !== 'https:') {
  return { valid: false, reason: 'Only HTTP and HTTPS protocols are allowed' };
}

// Extract hostname
const hostname = url.hostname;

// Check against whitelist
const isWhitelisted = ALLOWED_DOMAINS.some(domain => {
  return hostname === domain || hostname.endsWith(`.${domain}`);
});

if (!isWhitelisted) {
  return { valid: false, reason: 'Domain not in whitelist' };
}

// Resolve IP address
try {
  const { address } = await dnsLookup(hostname);

  // Check if IP is in forbidden range
  for (const range of FORBIDDEN_RANGES) {
    if (ipRangeCheck(address, range)) {
      return { valid: false, reason: 'IP address in forbidden range' };
    }
  }
} catch (dnsError) {
```

```
      return { valid: false, reason: 'Failed to resolve hostname' };
    }

    return { valid: true };
  } catch (error) {
    return { valid: false, reason: 'Invalid URL format' };
  }
}

// Proxy service endpoint
app.get('/proxy', async (req, res) => {
  const url = req.query.url;

  if (!url) {
    return res.status(400).json({ error: 'URL parameter is required' });
  }

  // Validate URL
  const validation = await validateUrl(url);
  if (!validation.valid) {
    return res.status(403).json({ error: `Invalid URL: ${validation.reason}` });
  }

  try {
    // Make the request with strict timeout and size limits
    const response = await axios({
      method: 'get',
      url: url,
      timeout: 5000,
      maxContentLength: 5 * 1024 * 1024, // 5MB limit
      maxRedirects: 0, // No redirects allowed
```

```
    responseType: 'stream'
  });

  // Forward response headers
  Object.entries(response.headers).forEach(([key, value]) => {
   // Filter out potentially dangerous headers
   if (!['set-cookie', 'transfer-encoding'].includes(key.toLowerCase())) {
    res.setHeader(key, value);
   }
  });

  // Set appropriate content type
  res.setHeader('Content-Type', response.headers['content-type'] || 'application/octet-stream');

  // Stream the response
  response.data.pipe(res);
 } catch (error) {
  const statusCode = error.response ? error.response.status : 500;
  const errorMessage = error.response ? `Error: ${error.response.statusText}` : `Request failed:
${error.message}`;

  res.status(statusCode).json({ error: errorMessage });
 }
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
 console.log(`Secure proxy service running on port ${PORT}`);
});
```

Java URL Validation and Secure Client

java

```
import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.UnknownHostException;
import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

import org.apache.http.HttpResponse;
import org.apache.http.client.config.RequestConfig;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;

public class SecureUrlFetcher {

    // List of allowed domains
    private static final List<String> ALLOWED_DOMAINS = Arrays.asList(
            "api.trusted-service.com",
            "cdn.approved-domain.org"
    );

    // Regex patterns for detecting internal IPs
    private static final Pattern LOCALHOST_PATTERN = Pattern.compile(
            "^(localhost|127\\.|0\\.0\\.0\\.0|\\[::1\\]|\\[0:0:0:0:0:0:0:1\\]).*$",
            Pattern.CASE_INSENSITIVE
    );
```

```
private static final Pattern PRIVATE_IP_PATTERN = Pattern.compile(
    "^(10\\.\\d+\\.\\d+\\.\\d+|" +
    "172\\.(1[6-9]|2\\d|3[0-1])\\.\\d+\\.\\d+|" +
    "192\\.168\\.\\d+\\.\\d+|" +
    "169\\.254\\.\\d+\\.\\d+)$",
    Pattern.CASE_INSENSITIVE
);

/**
 * Comprehensive URL validation for SSRF prevention
 */
public static boolean isUrlSafe(String urlString) {
   try {
      // Basic URL validation
      URL url = new URL(urlString);

      // Protocol validation
      String protocol = url.getProtocol().toLowerCase();
      if (!protocol.equals("http") && !protocol.equals("https")) {
         System.out.println("Invalid protocol: " + protocol);
         return false;
      }

      // Get hostname
      String hostname = url.getHost();

      // Check for localhost patterns
      if (LOCALHOST_PATTERN.matcher(hostname).matches()) {
         System.out.println("Localhost detected: " + hostname);
         return false;
      }
```

```
// Validate port (optional)
int port = url.getPort();
if (port != -1 && (port < 80 || port > 10000)) {
   System.out.println("Port outside allowed range: " + port);
   return false;
}

// Domain whitelist check
boolean isAllowedDomain = false;
for (String domain : ALLOWED_DOMAINS) {
   if (hostname.equals(domain) || hostname.endsWith("." + domain)) {
      isAllowedDomain = true;
      break;
   }
}

if (!isAllowedDomain) {
   System.out.println("Domain not in whitelist: " + hostname);
   return false;
}

// Resolve IP addresses
try {
   InetAddress[] addresses = InetAddress.getAllByName(hostname);
   for (InetAddress address : addresses) {
      String ip = address.getHostAddress();

      // Check for private IP addresses
      if (PRIVATE_IP_PATTERN.matcher(ip).matches()) {
         System.out.println("Private IP detected: " + ip);
```

```
                return false;
            }

            // Check for loopback addresses
            if (address.isLoopbackAddress()) {
                System.out.println("Loopback address detected: " + ip);
                return false;
            }

            // Check for link local addresses
            if (address.isLinkLocalAddress()) {
                System.out.println("Link local address detected: " + ip);
                return false;
            }
          }
        } catch (UnknownHostException e) {
            System.out.println("Failed to resolve hostname: " + hostname);
            return false;
        }

        return true;

    } catch (MalformedURLException e) {
        System.out.println("Malformed URL: " + e.getMessage());
        return false;
    }
}

/**
 * Securely fetch content from a URL with proper safeguards
 */
```

```
public static String fetchUrl(String urlString) {
   if (!isUrlSafe(urlString)) {
      return "URL failed security validation";
   }

   CloseableHttpClient httpClient = null;
   try {
      // Configure request with timeouts and restrictions
      RequestConfig config = RequestConfig.custom()
            .setConnectTimeout(5000)
            .setConnectionRequestTimeout(5000)
            .setSocketTimeout(5000)
            .setRedirectsEnabled(false)  // Prevent redirects
            .setCircularRedirectsAllowed(false)
            .build();

      // Create HTTP client with configuration
      httpClient = HttpClientBuilder.create()
            .setDefaultRequestConfig(config)
            .setConnectionTimeToLive(10, TimeUnit.SECONDS)
            .build();

      // Create and execute request
      HttpGet request = new HttpGet(urlString);
      HttpResponse response = httpClient.execute(request);

      // Check for redirects
      int statusCode = response.getStatusLine().getStatusCode();
      if (statusCode >= 300 && statusCode < 400) {
         return "Redirects are not allowed";
      }
```

```
        // Limit response size (5MB)
        int maxSize = 5 * 1024 * 1024;
        if (response.getEntity().getContentLength() > maxSize) {
            return "Response too large";
        }

        // Process response
        String responseBody = EntityUtils.toString(response.getEntity());
        return responseBody;

    } catch (IOException e) {
        return "Error fetching URL: " + e.getMessage();
    } finally {
        if (httpClient != null) {
            try {
                httpClient.close();
            } catch (IOException e) {
                // Ignore close errors
            }
        }
    }
}

public static void main(String[] args) {
    // Example usage
    String url = "https://api.trusted-service.com/data";
    if (isUrlSafe(url)) {
        String content = fetchUrl(url);
        System.out.println("Content: " + content);
    } else {
```

```
            System.out.println("URL failed security validation");
        }
    }
}
```