Understanding Authenticated HTTP PUT: From Binary Upload to a Windows Reverse Shell

HTTP `PUT` is not a vulnerability. Authentication is not a vulnerability either. The interesting security problem appears when several individually understandable behaviors compose into something the system's designers did not intend:

1. an authenticated identity can write request-body bytes;
2. the file service maps a URL to an unsafe filesystem location;
3. that location overlaps a directory used by a web server or another execution mechanism;
4. an uploaded file is interpreted as code, or is started by code that already executes; and
5. the resulting process can connect out to an operator-controlled listener.

Only the first item is really about `PUT`. The rest concern authorization, path canonicalization, filesystem boundaries, server-side execution, Windows process creation, and network egress. Keeping those concepts separate makes the attack chain easier to understand, test, report, and prevent.

> **Authorized environments only**
>
> The examples below are for an isolated lab or an explicitly authorized assessment. They use documentation-only IP ranges and placeholder credentials. Do not reproduce them against a third-party service.

The reference architecture

Consider two HTTP-speaking services on the same Windows host:

```
                          Windows host: TARGET

client  -- HTTPS/443 --> authenticated file service
                               |
                               | maps a URL to a filesystem path
                               v
                          shared storage
                               ^
                               | reads files from a webroot
                               |

client  <-- HTTP/80  --- Apache + PHP


Windows host  -- outbound TCP --> listener at LHOST:LPORT
```

The file service and Apache have different jobs. The first accepts authenticated file operations. The second serves web content and may pass selected file types to an interpreter such as PHP. They become one security problem only if their filesystem namespaces overlap.

That overlap is the architectural hinge. If the file service is confined to `C:\Data\Uploads` and Apache serves only `C:\Web\Public`, a successful upload remains data. If unsafe path handling lets the file service write into `C:\Web\Public`, the same upload can change what Apache serves. If PHP is enabled in that directory, a writable file can also become server-side code.

This is why the most useful question is not simply “Does the server allow uploads?” It is:

> Which authenticated identity can cause which exact bytes to be written to which canonical filesystem object, and what software will later consume that object?

What authenticated PUT actually establishes

At the HTTP layer, `PUT` asks a server to make the target resource reflect the representation in the request body. A simplified binary upload looks like this:

```
PUT /uploads/tool.bin HTTP/1.1
Host: 192.0.2.10
Authorization: Basic <BASE64_CREDENTIALS>
Content-Type: application/octet-stream
Content-Length: <BYTE_COUNT>

<RAW BINARY BYTES>
```

Each part answers a different question:

Message element	Meaning	What it does not prove
`PUT`	The client requests replacement or creation of a resource.	That the server permits the operation.
Request target	Names the resource in URL space.	The final Windows filesystem path.
`Authorization`	Presents credentials or an authorization token.	That authorization is correctly scoped.
`Content-Type`	Describes the body as arbitrary bytes.	That the server will execute those bytes.
`Content-Length`	States how many body bytes follow.	That the stored copy is complete or unchanged.
Request body	Carries the representation to store.	Where the server stores it.

Authentication establishes an identity. Authorization decides what that identity may do. A server can authenticate a user correctly and still authorize an unsafe destination. Conversely, a `401 Unauthorized` normally means the authentication exchange is incomplete, while a `403 Forbidden` usually means the server recognized the request but refuses the operation.

A successful response also needs interpretation. `201 Created` commonly means the server created a new resource; `200 OK` or `204 No Content` can indicate that an existing resource was updated. None of these status codes

independently proves where the bytes landed on disk. That requires an observation from the storage or consuming side of the system.

How curl turns a local file into an HTTP body

The following variables keep an authorized test reproducible without embedding a real target or password:

```
export TARGET=192.0.2.10
export LHOST=198.51.100.20
export USERNAME=labuser
export PASSWORD='<LAB_PASSWORD>'

export LPORT=4444
export PUT_PATH='<LAB_AUTHORIZED_UPLOAD_PATH>'
```

An illustrative binary `PUT` is:

```
curl --insecure \
  --request PUT \
  --user "${USERNAME}:${PASSWORD}" \
  --data-binary @./tool.bin \
  --path-as-is \
  "https://${TARGET}${PUT_PATH}/tool.bin"
```

This command matters because it controls four distinct layers:

`--request PUT` selects the HTTP method

The method is part of the server's authorization decision. A route might allow `GET` but reject `PUT` with `405 Method Not Allowed`; another might allow `PUT` only for authenticated principals or particular path prefixes.

`--data-binary @./tool.bin` supplies the representation

The leading `@` tells curl to read the named file and send its contents. Without it, curl sends the literal characters `./tool.bin`:

```
--data-binary @./tool.bin   -> bytes read from the file
--data-binary ./tool.bin    -> the text "./tool.bin"
```

`--data-binary` preserves the body as supplied. That is essential for executable files, archives, and other formats in which a changed byte can invalidate the result. The receiving service may still transform or reject the representation, so a successful request is not an integrity check.

`--user` constructs an authentication exchange

With HTTP Basic authentication, curl derives an `Authorization: Basic ...` header from the username and password. Base64 is an encoding, not encryption. TLS protects the header while it crosses the network, but terminal history, verbose logs, proxy projects, and screenshots can still expose it at either endpoint.

`--path-as-is` preserves curl's outgoing path

Clients commonly normalize dot segments in URLs before sending them. `--path-as-is` tells curl not to perform that client-side cleanup. It is useful when an authorized test needs to observe how the server itself handles an unusual request target.

It does **not** guarantee that the wire representation becomes the final storage path. A reverse proxy, framework, file-service router, or Windows API may decode and normalize the path later. It also does not disable URL encoding or change how a server treats `%2e`, `%2f`, backslashes, Unicode, or repeated separators.

`--insecure` changes TLS verification, not HTTP

This option allows a connection to a lab service whose certificate is not trusted by the client. It does not become an HTTP header and it does not weaken the server's authorization logic. It should not be a default for production testing because it removes certificate and hostname verification.

For diagnostics, `--verbose` shows the request and connection exchange, while `--include` includes response headers. Use them carefully: verbose output can contain reusable credentials.

The URL-to-filesystem boundary

The request target is a URL, not a Windows path. A typical service processes it through several transformations:

```
wire request target
  -> URL parsing
  -> percent decoding
  -> separator handling
  -> dot-segment normalization
  -> route or virtual-directory mapping
  -> storage-root joining
  -> canonical filesystem path
  -> authorization check

  -> file operation
```

The order matters. A robust service authorizes the final canonical object it will open. A vulnerable design may authorize an earlier string—perhaps one that appears to begin with an allowed directory—and then normalize it into a different destination before writing.

Windows adds several path representations that software must treat consistently: drive-letter paths, UNC paths, forward and backward slashes, junctions, case-insensitive names, and multiple textual spellings of the same location. A web framework and the underlying filesystem API may not canonicalize those representations in exactly the same way.

The general security invariant is simple:

> The path checked by the authorization policy must identify the same filesystem object that the write operation ultimately opens.

Path traversal is one way to violate that invariant, but it is not the only one. Unsafe virtual-directory mappings, writable junction targets, overly broad service-account permissions, and a deliberately exposed WebDAV-style root can create equivalent outcomes without classic `../` traversal.

A file write is not code execution

A file-write primitive answers “Can these bytes be stored here?” Remote code execution answers “Will a process interpret or load these bytes as instructions?” There is always a bridge between the two.

Observed behavior	Security capability established
The `PUT` receives a success response.	The service accepted the request.
A later `GET` returns the same bytes.	The write overlaps a readable resource mapping.
A script returns the result of a harmless expression.	A server-side interpreter is active for that file.
A child process appears under the web-service identity.	The interpreted code can create operating-system processes.
An outbound session reaches a listener.	Process creation and the callback network path both work.

Requesting an uploaded `.exe` through Apache normally downloads or serves the file; it does not ask Windows to launch it. `Content-Type: application/octet-stream` also has no execution semantics. The browser and server treat it as a representation, not as a process.

In an Apache/PHP architecture, a PHP file can provide the missing bridge because Apache sends it to the PHP runtime. If that script passes attacker-controlled input to a process-creation function, the interpreter can start a Windows program. The dangerous data flow is:

```
HTTP parameter
  -> PHP request parsing
  -> process-execution function
  -> Windows command-line parsing
  -> child process
```

This is conceptually a second vulnerability: arbitrary command execution in server-side code. The file-write issue makes it reachable by placing the script in an interpreted directory. If PHP is disabled, the file may be served as source or treated as static content. If the upload directory is configured as non-executable, the chain stops even though the file remains writable and web-accessible.

Other systems provide different bridges—deployment hooks, scheduled jobs, service configuration, plugin loaders, startup folders, or application-specific importers. A good analysis names the actual bridge instead of treating “upload” and “execution” as synonyms.

What a Windows reverse shell adds to the chain

A reverse shell is a process and network behavior, not a special property of HTTP. The target initiates an outbound connection to a waiting system and connects a command interpreter's input and output to that socket.

In the reference architecture, the process relationship is approximately:

```
Apache/PHP service identity
  -> command-execution bridge
    -> callback program
      -> cmd.exe or PowerShell
        <-> outbound TCP connection
          <-> listener at LHOST:LPORT
```

Several independent conditions must hold:

- **The path must resolve.** Windows must find the callback program from the service's working directory or from an absolute path.
- **The process must be permitted to start.** Filesystem ACLs, application control, antivirus, and EDR can block or terminate it.
- **The architecture must be compatible.** A malformed or incompatible executable fails before networking begins.
- **`LHOST` must be reachable from the target.** A VPN address, container address, NAT-only interface, or loopback address may be correct on the operator's machine but unreachable from the Windows host.
- **The listener must bind before the trigger.** Nothing is available to accept the connection otherwise.
- **Outbound traffic must be allowed.** Host firewalls, network ACLs, proxies, and egress filtering can stop the callback after code execution has already succeeded.
- **Standard handles must be connected correctly.** An outbound TCP connection alone is not an interactive shell; the child process's stdin, stdout, and usually stderr must be bridged to it.

The shell runs as the identity of the web-service process unless another privilege boundary is crossed. Valid upload credentials do not become Windows credentials, and obtaining a callback does not imply administrative access.

Why the HTTP trigger can appear to hang

Many server-side process functions are synchronous. If a PHP request starts a callback program and waits for it to exit, the HTTP response remains open for the lifetime of that process. The browser, curl, or Burp Repeater may therefore show a pending request while the listener has an active session.

That behavior is not proof by itself: a request can hang for unrelated reasons. It is simply consistent with a long-lived child process. The listener state, server logs, and process telemetry provide stronger evidence.

Encoding crosses several parsers

A command delivered through an HTTP parameter passes through more than one grammar:

```
local shell
  -> curl argument parsing
  -> URL encoding
  -> HTTP request-target parsing
  -> application query parsing
  -> PHP string handling
  -> Windows command-line parsing
```

A character can be ordinary data at one layer and syntax at the next. Spaces, `&`, `%`, `+`, backslashes, quotes, and redirection characters are common sources of confusion.

For an intentionally vulnerable lab handler, curl's `--data-urlencode` is clearer than manually assembling a query string:

```
curl --get \
  --data-urlencode 'cmd=whoami' \
  --url "http://${TARGET}/cmd.php"
```

Here, `--get` moves the data into the query string and `--data-urlencode` encodes the parameter value. This solves URL construction; it does not solve Windows quoting inside the receiving application. It is also separate from `--path-as-is`, which affects curl's handling of the URL path rather than the contents of a query parameter.

Reading the same request in Burp Repeater

curl is useful for repeatability. Burp Repeater is useful for separating the connection destination from the HTTP message and examining the response after each change.

The binary `PUT` has the same conceptual structure in either tool:

```
PUT /<LAB_AUTHORIZED_UPLOAD_PATH>/tool.bin HTTP/1.1
Host: 192.0.2.10
Authorization: Basic <BASE64_CREDENTIALS>
Content-Type: application/octet-stream
Connection: close

<RAW BINARY BYTES>
```

curl concept	Repeater concept
`https://TARGET`	Destination host, port `443`, and TLS enabled
`--request PUT`	Method in the request line
`--user ...`	`Authorization` header
`--data-binary @FILE`	Raw message body loaded from a file
`--path-as-is`	Literal request target retained in the editor
curl's body sizing	Repeater recalculates `Content-Length`

Binary data should be loaded directly into the message body. Copying an executable through a text clipboard can alter bytes or truncate at control characters. Repeater should also be allowed to update `Content-Length` after the body changes.

The destination controls which network service receives the request; the `Host` header participates in HTTP virtual hosting. They are related but not interchangeable. Sending an upload-shaped request to Apache on port `80` can produce an Apache error even when the intended file service on `443` would accept it. That is a routing mistake, not evidence about the file service's authorization.

Build evidence one boundary at a time

The cleanest analysis treats the chain as a series of claims, each with its own evidence:

Claim	Useful evidence	What remains unknown
The identity may use `PUT`.	Redacted request and successful response.	Final storage destination.
The intended bytes were stored.	Retrieval plus matching cryptographic hashes.	Whether any runtime interprets them.
The web mapping reaches the file.	`HEAD` / `GET`, server logs, or local filesystem evidence.	Server-side execution.
A script interpreter is active.	Harmless deterministic output from the interpreter.	OS command execution.
Commands run under a service identity.	`whoami`, process telemetry, and parent-child relationship.	Outbound reachability.
A callback path works.	Listener connection plus network/process telemetry.	Privileges beyond the service identity.

Matching hashes are especially useful for binary uploads:

```
curl --silent --show-error \
  "http://${TARGET}/tool.bin" \
  --output /tmp/tool.remote.bin

sha256sum ./tool.bin /tmp/tool.remote.bin
```

A failed claim narrows the problem. If a benign command executes but no callback arrives, there is little value in repeating the upload: the file-write and execution boundaries have already been established. Attention should move to executable path resolution, process controls, listener binding, routing, and egress policy.

Status codes are clues, not conclusions

Observation	Common interpretation
`401`	Credentials are absent, invalid, or not accepted by the advertised scheme.
`403`	The request is understood but the authenticated identity is not allowed to perform it.
`404` after a successful write	The web mapping and write destination may not overlap.
`405`	The selected endpoint does not allow `PUT`.
`409`	The server cannot create the resource in the current hierarchy or state.
PHP source is returned	The file is being served statically rather than interpreted.
A harmless command works but the callback does not	Execution works; process startup or outbound networking remains unproven.
The trigger stays pending	A synchronous child process is possible; inspect the listener and process state.

These are starting hypotheses. Server logs and direct evidence should decide the diagnosis.

Why this chain matters defensively

The strongest mitigation is architectural separation. No single parser or access-control check should be responsible for containing the entire chain.

- Disable `PUT` where the application does not require it.
- Authenticate users with an appropriate mechanism, but treat authentication as only the start of authorization.
- Canonicalize and decode a path before authorizing the final filesystem object.
- Reject paths that escape the configured storage root after canonicalization.
- Run the file service and web server under separate, least-privileged identities.
- Keep writable storage outside every executable webroot.

- Disable script handlers and execution permission in user-writable directories.
- Use allow-listed extensions and content validation as additional controls, not as the primary boundary.
- Restrict outbound traffic from web-server processes to necessary destinations and ports.
- Apply application control and endpoint monitoring to unexpected child processes.
- Protect proxy histories, shell histories, and debug logs as credential-bearing records.

Detection should correlate events across layers. A `PUT` request is often legitimate in isolation. A stronger signal is an unusual `PUT` followed by a new script in a webroot, a child process spawned by Apache or PHP, and an outbound connection from that process.

Useful telemetry includes:

- web access logs with method, normalized path, authenticated identity, status, and response size;
- file-service logs that record the final canonical destination;
- filesystem auditing for new executable or script files in served directories;
- process creation showing the web-service parent and command-line arguments; and
- network telemetry for unexpected egress from the web-service identity.

How to describe the finding accurately

“Unrestricted file upload” is incomplete if authentication, unsafe path resolution, and server-side interpretation are all necessary. “Reverse shell vulnerability” is also imprecise because the shell is an outcome, not the root cause.

A defensible report separates the chain:

1. the authenticated principal and permitted HTTP method;
2. the path-mapping or authorization weakness;
3. the canonical write destination;
4. the overlap with an interpreted or otherwise executable location;
5. the process identity and demonstrated command execution; and
6. the outbound network condition that permitted interactive access.

The report should also say which links were observed and which were inferred. Redact passwords and authorization headers, use harmless commands for proof, avoid publishing target-specific paths, and document cleanup of every uploaded test artifact.

The compact mental model

Authentication identifies the caller. `PUT` carries bytes. Path mapping decides where those bytes land. An execution bridge turns stored data into a process. Windows starts that process under a specific identity. A reverse shell exists only when the process can connect out and attach a command interpreter to the socket.

Return to **Offensive Security Blog**.