

COMMONWEALTH OF MASSACHUSETTS

MIDDLESEX, SS.

SUPERIOR COURT

BO SHANG,

Plaintiff,

v.

MIDDLESEX COUNTY DISTRICT

ATTORNEY'S OFFICE,

Defendant.

COMPLAINT AND JURY DEMAND (AS ENHANCED)

Plaintiff, Bo Shang ("Plaintiff"), brings this Complaint against Defendant Middlesex County District Attorney's Office ("Defendant") and alleges as follows, incorporating additional factual and legal authorities:

1 Plaintiff is an individual residing in 10 McCafferty Way, Burlington MA 01803.

2 Defendant is a public office located in Middlesex County, Massachusetts.

JURISDICTION AND VENUE

3 This Court has subject matter jurisdiction pursuant to G.L. c. 212, § 4, and under concurrent jurisdiction principles for claims brought under 42 U.S.C. § 1983. See Haywood v. Drown, 556 U.S. 729 (2009). This Court also has jurisdiction over claims

arising under the Massachusetts Constitution, the Massachusetts Civil Rights Act (MCRA), G.L. c. 12, §§ 11H & 11I, and other Massachusetts common law claims.

4 Venue is proper in this Court pursuant to G.L. c. 223, § 1, because the events or omissions giving rise to this action occurred in Middlesex County and because Defendant is located in Middlesex County.

FACTUAL BACKGROUND

5 On or about January 8, 2025, Plaintiff alleges that the Middlesex District Attorney's Office of MA, described by Plaintiff as "corrupt and despicably morally principled," filed a data request to Apple, supposedly under Massachusetts Rules of Civil Procedure 45.

6 This occurred on the same day Plaintiff filed a motion to dismiss Twitch and an AirTag + commerce tax (App Store) lawsuit against Apple (which hosts the Twitch app).

7 Plaintiff asserts that this data request was not legitimately obtained under Mass. R. Civ. P. 45 but was instead an illegal measure taken against Plaintiff as an "enemy combatant," contrary to both domestic and international law, including Geneva Conventions III & IV, and the International Covenant on Civil and Political Rights

(ICCPR).

7A. The United States is a party to the four Geneva Conventions of 1949, which set forth standards for treatment of persons in armed conflicts, including alleged "enemy combatants." Plaintiff maintains that labeling Plaintiff as an "enemy combatant" without due process violates customary international humanitarian law and Supreme Court precedent concerning the rights of such individuals. See, e.g., Hamdi v. Rumsfeld, 542 U.S. 507 (2004); Rasul v. Bush, 542 U.S. 466 (2004); Boumediene v. Bush, 553 U.S. 723 (2008).

7B. The United States is also a State Party to the ICCPR, which, under Article 9, protects against arbitrary arrest or detention and, under Article 14, protects due process rights. Plaintiff alleges that classifying Plaintiff as an "enemy combatant" in a civilian context, and thereby circumventing ordinary legal process, violates the ICCPR's guarantees of

73 fundamental procedural protections.

74

75 7C. The Supreme Court has further clarified the rights of individuals designated as “enemy
76 combatants” in *Padilla v. Rumsfeld*, 542 U.S. 426 (2004), emphasizing the need for
77 proper legal process. Plaintiff alleges these precedents reinforce the argument that
78 civilian processes cannot be bypassed via “enemy combatant” designations.

79

80 7D. In *Ex parte Milligan*, 71 U.S. (4 Wall.) 2 (1866), the Supreme Court held that applying
81 military or martial process to civilians, when civil courts are open, is unconstitutional.
82 Plaintiff contends this principle applies here, making any civilian “enemy combatant”
83 label unlawful.

84

85 7E. The Supreme Court in *Hamdan v. Rumsfeld*, 548 U.S. 557 (2006), further confirmed
86 that efforts to circumvent civilian courts through alternative proceedings for alleged
87 combatants violate U.S. constitutional principles. Plaintiff alleges that all such
88 precedents collectively prohibit unilateral “enemy combatant” branding in non-war
89 contexts.

90

91 **8 On January 30, 2025, Plaintiff received an email from Apple regarding this request,**
92 **which stated in part:**

93

94 “Apple

95

96 **NOTE: THIS NOTICE IS BEING SENT FROM A NO-REPLY EMAIL ACCOUNT—ANY RESPONSE**

97

98 **TO THIS EMAIL WILL NOT RECEIVE A RESPONSE**

99

100 Dear Account Holder/Customer:

101

102 On 2025-01-08, Apple Inc. (“Apple”) received a legal request from Middlesex District
103 Attorney's Office requesting information regarding your Apple account.

104

105 The contact information in relation to the request:

106 Requesting Agency: Middlesex District Attorney's Office

107 Requesting Agency Location: Woburn, MA - Massachusetts

108 Requesting Agency Case Number: 2024-398

109 Legal Request Type: Subpoena / Summons

110

111 Pursuant to the applicable Terms of Service and Apple's Privacy Policy,

112 <http://www.apple.com/legal/privacy/en-ww/>, and as required by U.S. law, Apple

113 will be producing the requested data in a timely manner as required by the legal

114 process. If you have questions about the legal request or the information requested,

115 please contact the requesting agency.

116

117 Sincerely,

118 Apple Privacy & Law Enforcement Compliance

119 Apple Inc."

120

121 **9 Plaintiff maintains that Defendant violated Plaintiff's rights under federal and state law**

122 by improperly obtaining and misusing personal data. Plaintiff asserts a violation of

123 privacy rights under G.L. c. 214, § 1B (right against unreasonable, substantial or

124 serious interference with privacy), Article 14 of the Massachusetts Declaration of

125 Rights (protection against unreasonable searches and seizures), the Fourth Amendment

126 to the U.S. Constitution, and international human rights norms including Article 17 of

127 the ICCPR and Article 12 of the Universal Declaration of Human Rights (UDHR).

128

129 9A. The UDHR, though not a binding treaty, informs customary international law and reflects

130 global human rights standards. Article 12 states that "[n]o one shall be subjected to

131 arbitrary interference with his privacy," a principle Plaintiff contends was violated.

132

133 9B. The United States is also a State Party to the Convention Against Torture (CAT),

134 highlighting due process norms. Plaintiff claims that Defendant's labeling and treatment

135 of Plaintiff as an "enemy combatant" violate the spirit of these international

136 commitments.

137

138 9C. In *United States v. Warshak*, 631 F.3d 266 (6th Cir. 2010), the court recognized a

139 reasonable expectation of privacy in certain electronic communications, requiring

140 proper legal process for data access. Plaintiff alleges Defendant's conduct flouts

141 Warshak's privacy rationale.

142

143 9D. In *Kyllo v. United States*, 533 U.S. 27 (2001), the Supreme Court held that obtaining

144 information through technology not otherwise accessible without physical intrusion

145 implicates the Fourth Amendment. Plaintiff characterizes Defendant's subpoena or
146 data request as an analogous overreach.

147
148 9E. Under Massachusetts jurisprudence, the Supreme Judicial Court in Commonwealth v.
149 Augustine, 467 Mass. 230 (2014), recognized strong privacy protections for personal
150 digital records, requiring heightened procedures for obtaining certain data. Plaintiff
151 alleges that Defendant's conduct runs afoul of Augustine's reasoning.

152
153 **10 Plaintiff alleges that, in response to Defendant's perceived threat, Plaintiff invoked the**
154 Second Amendment to the U.S. Constitution, as recognized in District of Columbia
155 v. Heller, 554 U.S. 570 (2008), McDonald v. City of Chicago, 561 U.S. 742 (2010), and
156 Caetano v. Massachusetts, 577 U.S. 411 (2016). Plaintiff also invokes Article 17 of
157 the Massachusetts Declaration of Rights, contending these decisions protect an
158 individual right to bear "arms," which Plaintiff interprets to include "cyber arms."

159
160 **11 Plaintiff claims to have developed or acquired "cyber arms" by creating advanced**
161 persistent threats ("APTs") and by allying with other APTs, including "Salt Typhoon."
162 Plaintiff asserts that these "cyber arms" are protected under the Second Amendment
163 and Article 17 as a form of self-defense.

164
165 **12 Plaintiff alleges that Defendant's conduct in issuing or causing the issuance of a data**
166 request without valid legal basis constituted an unlawful intrusion upon Plaintiff's data
167 privacy, in violation of the Fourth Amendment (as incorporated by Mapp v. Ohio, 367
168 U.S. 643 (1961), and recognized in Katz v. United States, 389 U.S. 347 (1967), Terry v.
169 Ohio, 392 U.S. 1 (1968), Carpenter v. United States, 138 S. Ct. 2206 (2018), Riley v.
170 California, 573 U.S. 373 (2014)), Article 14 of the Massachusetts Declaration of Rights,
171 the Stored Communications Act (18 U.S.C. §§ 2701–2712), Article 17 of the ICCPR,
172 and Article 12 of the UDHR.

173
174 12A. Plaintiff notes that third-party data requests implicate the "third-party doctrine," as set
175 forth in Smith v. Maryland, 442 U.S. 735 (1979). However, Carpenter recognized
176 limitations when sensitive digital data is at issue. Plaintiff alleges that Defendant's
177 conduct violates Carpenter's narrowing of the third-party doctrine.

178
179 12B. Plaintiff further cites Commonwealth v. Gouse, 461 Mass. 787 (2012), for the
180 proposition that Massachusetts courts often apply heightened scrutiny to searches

181 involving personal or digital privacy, reinforcing Plaintiff's claim that Defendant's
182 subpoena was invalid or overreaching.

183
184 **13 Plaintiff contends that Defendant's conduct effectively labeled Plaintiff an "enemy**
185 **combatant,"** heightening constitutional concerns, implicating Article 5 of the UDHR, and
186 prompting Plaintiff's reliance on the Second Amendment and Article 17 to protect
187 "cyber arms" from confiscation, regulation, or direct infringement.

188
189 13A. Plaintiff invokes *Hamdan v. Rumsfeld*, 548 U.S. 557 (2006), to underscore the illegality
190 of any extrajudicial designation of "enemy combatant" status. Plaintiff argues that
191 under both domestic and international law, such designations cannot bypass civilian
192 jurisdiction in ordinary contexts.

193
194 **14 Plaintiff asserts that Defendant's actions violate customary international law norms**
195 **related to privacy,** as recognized by multiple treaties and conventions to which the
196 United States is a party or signatory, including the ICCPR, and contravene prohibitions
197 on arbitrary interference under global human rights standards.

198
199 14A. The United States is a signatory to the Budapest Convention on Cybercrime, addressing
200 lawful cooperation in criminal cyber matters. Plaintiff contends that Defendant's
201 allegedly improper "cyber" classification and data request contravene the spirit of
202 privacy protections contemplated by such instruments.

203
204 14B. Although the United States has not ratified Additional Protocol I or II to the Geneva
205 Conventions, Plaintiff argues that certain principles therein reflect customary
206 international humanitarian law, prohibiting arbitrary or extrajudicial designations
207 of civilians as combatants.

208
209 14C. The United States is also a member of the Organization of American States and is bound
210 by certain obligations under the American Declaration of the Rights and Duties of Man,
211 which can inform interpretations of privacy and due process in conjunction with other
212 international norms.

213
214 14D. In addition, *N.Y. State Rifle & Pistol Assn. v. Bruen*, 597 U.S. ____ (2022), further
215 clarified the scope of the Second Amendment right to bear arms. Plaintiff references
216 *Bruen* to argue that Defendant's attempts to limit, seize, or regulate "cyber arms"

are inconsistent with the broad individual right recognized by the Supreme Court.

CAUSES OF ACTION

COUNT I

(Violation of 42 U.S.C. § 1983)

15 Plaintiff repeats and re-alleges all preceding paragraphs as though fully set forth herein.

16 Defendant, acting under color of state law, allegedly caused the issuance of a subpoena or summons without proper legal basis in violation of Plaintiff's constitutional rights, including but not limited to the Fourth Amendment right to be free from unreasonable searches and seizures as recognized in Katz, Terry, Mapp, Carpenter, Riley, and related precedent.

17 By issuing or causing this allegedly improper process, Defendant deprived Plaintiff of rights secured by the Constitution and laws of the United States, in contravention of

42 U.S.C. § 1983.

COUNT II

(Violation of Massachusetts Civil Rights Act)

18 Plaintiff repeats and re-alleges all preceding paragraphs as though fully set forth herein.

19 Defendant's conduct—issuing a data request under color of law without legitimate basis—constitutes interference or attempted interference with Plaintiff's exercise or enjoyment of rights secured by the Constitutions and laws of the United States and the Commonwealth, including the right against unreasonable searches (Article 14) and the right to keep arms (Article 17), by means of threats, intimidation, or coercion, in violation of G.L. c. 12, §§ 11H & 11I. See Batchelder v. Allied Stores Int'l, Inc., 388 Mass. 83 (1983); Buster v. George W. Moore, Inc., 438 Mass. 635 (2003); Commonwealth v. Powell, 459 Mass. 572 (2011).

20 As a direct and proximate result of Defendant's actions, Plaintiff has suffered and will continue to suffer damages recoverable under the MCRA.

253

254

COUNT III

255 (Abuse of Process Under Massachusetts Law)

256

257 **21 Plaintiff repeats and re-alleges all preceding paragraphs as though fully set forth herein.**

258

259 **22 Under Massachusetts law, an abuse of process claim arises when legal process is used**

260 for an ulterior or illegitimate purpose. See Cohen v. Hurley, 20 Mass. App. Ct. 439

261 (1985); Kelley v. Stop & Shop Cos., 26 Mass. App. Ct. 557 (1988); Lorusso v. Bloom,

262 321 Mass. 9 (1947).

263

264 **23 Defendant allegedly misused legal process by pursuing a data request unsupported by**

265 valid legal grounds and did so for an improper purpose, causing harm to Plaintiff.

266

267 **24 As a direct and proximate result of Defendant's actions, Plaintiff has suffered damages**

268 recoverable under Massachusetts law.

269

270 **COUNT IV**

271 (Injunctive Relief Under Federal and State Law)

272

273 **25 Plaintiff repeats and re-alleges all preceding paragraphs as though fully set forth herein.**

274

275 **26 As a result of Defendant's conduct, Plaintiff seeks injunctive relief prohibiting**

276 Defendant from further unlawful use of subpoenas, summonses, or other legal process

277 to access Plaintiff's personal data without proper justification. Plaintiff seeks to enjoin

278 any acts by Defendant that violate Plaintiff's rights under federal and state law,

279 including the Fourth Amendment, Article 14, G.L. c. 214, § 1B, the MCRA, the Stored

280 Communications Act, and international human rights treaties such as the ICCPR.

281

282 **COUNT V**

283 (Assertion of the Second Amendment and

284 Article 17 of the Massachusetts Declaration of Rights)

285

286 **27 Plaintiff repeats and re-alleges all preceding paragraphs as though fully set forth herein.**

287

288 **28 The Second Amendment states that "the right of the people to keep and bear Arms,**

shall not be infringed.” As held in *District of Columbia v. Heller*, 554 U.S. 570 (2008), this right is individual in nature, and in *McDonald v. City of Chicago*, 561 U.S. 742 (2010), it applies to the states. In *Caetano v. Massachusetts*, 577 U.S. 411 (2016), the Supreme Court reiterated its broad scope. Article 17 of the Massachusetts Declaration of Rights similarly protects the right to keep and bear arms. *N.Y. State Rifle & Pistol Assn. v. Bruen*, 597 U.S. ____ (2022), further refines these constitutional principles.

29 Plaintiff asserts that “cyber arms” (i.e., advanced persistent threats, digital tools, or alliances with groups such as “Salt Typhoon”) constitute protected “arms” under the Second Amendment and Article 17. Plaintiff alleges that any attempt by Defendant to seize, regulate, or otherwise interfere with these “cyber arms” without due process violates Plaintiff’s federal and state constitutional rights.

30 Plaintiff further alleges that Defendant’s labeling of Plaintiff as an “enemy combatant” or any related act to disarm Plaintiff’s “cyber capacity” contravenes *Heller*, *McDonald*, *Caetano*, *Bruen*, and Article 17 of the Massachusetts Declaration of Rights.

31 Plaintiff therefore seeks declaratory relief that any effort by Defendant to restrict Plaintiff’s possession or development of “cyber arms” violates the Second Amendment and Article 17, and that such restriction contravenes self-defense principles acknowledged by various human rights instruments, including the UN Charter’s Article 51 (albeit in state contexts) and related customary international law.

REQUEST FOR RELIEF

WHEREFORE, Plaintiff respectfully requests that this Court:

- A. Enter judgment in favor of Plaintiff and against Defendant on all causes of action;
- B. Award Plaintiff compensatory, consequential, and punitive damages in an amount to be determined at trial;

C Grant injunctive relief restraining Defendant from seeking or using Plaintiff’s personal data without proper legal justification;

D Declare that Plaintiff’s “cyber arms” are protected under the Second Amendment and

Article 17, and that any attempt by Defendant to restrict or confiscate them, if any,
violates federal and state constitutions and relevant international human rights standards;
E. Award Plaintiff's reasonable attorneys' fees and costs pursuant to 42 U.S.C. § 1988,
G.L. c. 12, §§ 11H & 11I, or as otherwise provided by law;
F. Grant such other and further relief as the Court deems just and proper.

DEMAND FOR JURY TRIAL

Plaintiff demands a trial by jury on all issues so triable.

Dated: 2/27/2025

Respectfully submitted,

Bo Shang
10 McCafferty Way
Burlington MA 01803-3127
202-235-5017 | 781-999-4101
bo@shang.software | enigmatic.typhoon@gmail.com

361
362
363
364
365
366 **EXHIBIT 1: On 1/30/25, the Plaintiff receives an email from Apple detailing the information request made to**
367 **the Plaintiff's developer account on 1/8/25, by the Middlesex DA's Office. This date coincided with the**
368 **Plaintiff filing 27 and 27-1 in Federal Court Case 3:24-cv-06664-JS, the first time ever anyone has won a**
369 **Section 230 claim vs Twitch interactive.**
370
371 <https://www.fakeopenai.co/section230>
372
373 <https://www.fakeopenai.co/lsat>
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389 **EXHIBIT 2: The Plaintiff is making great progress, and expects to achieve an "Eternal" family of zero-day**
390 **capabilities on the SMBv2 protocol, within a day or few days.**
391
392 `/*****`
393 `* File: smb2_pipe_exec_client.c ss`
394 `*`
395 `* Demonstrates:`
396 `* 1. Connecting to an SMB2/3 server (TCP 445).`

397	* 2. Negotiate, Session Setup, Tree Connect to IPC\$.	397
398	* 3. Create/open the named pipe "\\PIPE\\svctl".	398
399	* 4. Partially demonstrate sending a DCERPC bind	399
400	* request to the SVCCTL interface (stub only).	400
401	* 5. Read back any server response.	401
402	* 6. Close the pipe with an SMB2 Close.	402
403	*	403
404	* Security & Production Warnings:	404
405	* - This remains incomplete demonstration code:	405
406	* - No real auth or signing.	406
407	* - No real DCERPC parse/marshalling logic.	407
408	* - Minimal error handling and no encryption.	408
409	* - Use only in a controlled environment with	409
410	* permission!	410
411	*****/	411
412		412
413	#include <stdio.h>	413
414	#include <stdlib.h>	414
415	#include <string.h>	415
416	#include <unistd.h>	416
417	#include <arpa/inet.h>	417
418	#include <stdint.h>	418
419	#include <errno.h>	419
420		420
421	#pragma pack(push, 1)	421
422		422
423	//-----	423
424	// SMB2 Header	424
425	//-----	425
426	typedef struct _SMB2Header {	426
427	unsigned char ProtocolId[4]; // 0xFE 'S' 'M' 'B'	427
428	uint16_t StructureSize; // Always 64 for SMB2	428
429	uint16_t CreditCharge; // Credits requested/charged	429
430	uint32_t Status; // For responses, server sets status	430
431	uint16_t Command; // SMB2 command code	431
432	uint16_t Credits; // Credits granted/requested	432

433	uint32_t Flags; // SMB2 header flags	433
434	uint32_t NextCommand; // Offset to next command in compound	434
435	uint64_t MessageId; // Unique message ID	435
436	uint32_t Reserved; // Usually 0	436
437	uint32_t TreeId; // Tree ID	437
438	uint64_t SessionId; // Session ID	438
439	unsigned char Signature[16]; // For signing (unused here)	439
440	} SMB2Header;	440
441		441
442	// SMB2 Commands	442
443	#define SMB2_NEGOTIATE 0x0000	443
444	#define SMB2_SESSION_SETUP 0x0001	444
445	#define SMB2_TREE_CONNECT 0x0003	445
446	#define SMB2_CREATE 0x0005	446
447	#define SMB2_CLOSE 0x0006	447
448	#define SMB2_READ 0x0008	448
449	#define SMB2_WRITE 0x0009	449
450		450
451	// SMB2 Status Codes (common)	451
452	#define STATUS_SUCCESS 0x00000000	452
453	#define STATUS_INVALID_PARAMETER 0xC000000D	453
454	#define STATUS_ACCESS_DENIED 0xC0000022	454
455	#define STATUS_NOT_SUPPORTED 0xC00000BB	455
456		456
457	// SMB2 Dialects	457
458	#define SMB2_DIALECT_0202 0x0202	458
459	#define SMB2_DIALECT_0210 0x0210	459
460	#define SMB2_DIALECT_0300 0x0300	460
461		461
462	//-----	462
463	// Minimal Structures for Basic SMB2 Ops	463
464	//-----	464
465		465
466	/* SMB2 NEGOTIATE */	466
467	typedef struct _SMB2NegotiateRequest {	467
468	uint16_t StructureSize; // Must be 36	468

469	uint16_t DialectCount;	469
470	uint16_t SecurityMode;	470
471	uint16_t Reserved;	471
472	uint32_t Capabilities;	472
473	uint64_t ClientGuid; // Simplified to 8 bytes for demonstration	473
474	uint32_t NegotiateContextOffset;	474
475	uint16_t NegotiateContextCount;	475
476	uint16_t Reserved2;	476
477	// Then dialect array	477
478	} SMB2NegotiateRequest;	478
479		479
480	typedef struct _SMB2NegotiateResponse {	480
481	uint16_t StructureSize; // Must be 65 in real SMB2	481
482	uint16_t SecurityMode;	482
483	uint16_t DialectRevision;	483
484	uint16_t NegotiateContextCount;	484
485	uint32_t ServerGuid; // Simplified	485
486	uint32_t Capabilities;	486
487	uint32_t MaxTransSize;	487
488	uint32_t MaxReadSize;	488
489	uint32_t MaxWriteSize;	489
490	uint64_t SystemTime;	490
491	uint64_t ServerStartTime;	491
492	// etc...	492
493	} SMB2NegotiateResponse;	493
494		494
495	/* SMB2 SESSION_SETUP */	495
496	typedef struct _SMB2SessionSetupRequest {	496
497	uint16_t StructureSize; // Must be 25	497
498	uint8_t Flags;	498
499	uint8_t SecurityMode;	499
500	uint32_t Capabilities;	500
501	uint32_t Channel;	501
502	uint16_t SecurityBufferOffset;	502
503	uint16_t SecurityBufferLength;	503
504	// Security buffer follows...	504

505	} SMB2SessionSetupRequest;	505
506		506
507	typedef struct _SMB2SessionSetupResponse {	507
508	uint16_t StructureSize; // Must be 9	508
509	uint16_t SessionFlags;	509
510	uint16_t SecurityBufferOffset;	510
511	uint16_t SecurityBufferLength;	511
512	// ...	512
513	} SMB2SessionSetupResponse;	513
514		514
515	/* SMB2 TREE_CONNECT */	515
516	typedef struct _SMB2TreeConnectRequest {	516
517	uint16_t StructureSize; // Must be 9	517
518	uint16_t Reserved;	518
519	uint32_t PathOffset;	519
520	uint32_t PathLength;	520
521	// Path follows	521
522	} SMB2TreeConnectRequest;	522
523		523
524	typedef struct _SMB2TreeConnectResponse {	524
525	uint16_t StructureSize; // Must be 16	525
526	uint8_t ShareType;	526
527	uint8_t Reserved;	527
528	uint32_t ShareFlags;	528
529	uint32_t Capabilities;	529
530	uint32_t MaximalAccess;	530
531	} SMB2TreeConnectResponse;	531
532		532
533	/* SMB2 CREATE */	533
534	typedef struct _SMB2CreateRequest {	534
535	uint16_t StructureSize; // Must be 57	535
536	uint8_t SecurityFlags;	536
537	uint8_t RequestedOplockLevel;	537
538	uint32_t ImpersonationLevel;	538
539	uint64_t SmbCreateFlags;	539
540	uint64_t Reserved;	540

541	uint32_t DesiredAccess;	541
542	uint32_t FileAttributes;	542
543	uint32_t ShareAccess;	543
544	uint32_t CreateDisposition;	544
545	uint32_t CreateOptions;	545
546	uint16_t NameOffset;	546
547	uint16_t NameLength;	547
548	uint32_t CreateContextsOffset;	548
549	uint32_t CreateContextsLength;	549
550	// Filename follows...	550
551	} SMB2CreateRequest;	551
552		552
553	typedef struct _SMB2CreateResponse {	553
554	uint16_t StructureSize; // Must be 89	554
555	uint8_t OplockLevel;	555
556	uint8_t Flags;	556
557	uint32_t CreateAction;	557
558	uint64_t CreationTime;	558
559	uint64_t LastAccessTime;	559
560	uint64_t LastWriteTime;	560
561	uint64_t ChangeTime;	561
562	uint64_t AllocationSize;	562
563	uint64_t EndOfFile;	563
564	uint32_t FileAttributes;	564
565	// 16-byte FileId	565
566	uint64_t FileIdPersistent;	566
567	uint64_t FileIdVolatile;	567
568	// optional create contexts	568
569	} SMB2CreateResponse;	569
570		570
571	/* SMB2 WRITE/READ (for the RPC data) */	571
572	typedef struct _SMB2WriteRequest {	572
573	uint16_t StructureSize; // Must be 49	573
574	uint16_t DataOffset;	574
575	uint32_t Length;	575
576	uint64_t Offset;	576

577	uint64_t FileIdPersistent;	577
578	uint64_t FileIdVolatile;	578
579	uint32_t Channel;	579
580	uint32_t RemainingBytes;	580
581	uint16_t WriteChannelInfoOffset;	581
582	uint16_t WriteChannelInfoLength;	582
583	uint32_t Flags;	583
584	// Then the data	584
585	} SMB2WriteRequest;	585
586		586
587	typedef struct _SMB2WriteResponse {	587
588	uint16_t StructureSize; // Must be 17	588
589	uint16_t Reserved;	589
590	uint32_t Count;	590
591	uint32_t Remaining;	591
592	uint16_t WriteChannelInfoOffset;	592
593	uint16_t WriteChannelInfoLength;	593
594	} SMB2WriteResponse;	594
595		595
596	typedef struct _SMB2ReadRequest {	596
597	uint16_t StructureSize; // Must be 49	597
598	uint8_t Padding;	598
599	uint8_t Reserved;	599
600	uint32_t Length;	600
601	uint64_t Offset;	601
602	uint64_t FileIdPersistent;	602
603	uint64_t FileIdVolatile;	603
604	uint32_t MinimumCount;	604
605	uint32_t Channel;	605
606	uint32_t RemainingBytes;	606
607	uint16_t ReadChannelInfoOffset;	607
608	uint16_t ReadChannelInfoLength;	608
609	} SMB2ReadRequest;	609
610		610
611	typedef struct _SMB2ReadResponse {	611
612	uint16_t StructureSize; // Must be 17	612

613	uint8_t DataOffset;	613
614	uint8_t Reserved;	614
615	uint32_t DataLength;	615
616	uint32_t DataRemaining;	616
617	uint32_t Reserved2;	617
618	// data follows	618
619	} SMB2ReadResponse;	619
620		620
621	/* SMB2 CLOSE */	621
622	typedef struct _SMB2CloseRequest {	622
623	uint16_t StructureSize; // Must be 24	623
624	uint16_t Flags;	624
625	uint32_t Reserved;	625
626	uint64_t FileIdPersistent;	626
627	uint64_t FileIdVolatile;	627
628	} SMB2CloseRequest;	628
629		629
630	typedef struct _SMB2CloseResponse {	630
631	uint16_t StructureSize; // Must be 60	631
632	uint16_t Flags;	632
633	uint32_t Reserved;	633
634	uint64_t CreationTime;	634
635	uint64_t LastAccessTime;	635
636	uint64_t LastWriteTime;	636
637	uint64_t ChangeTime;	637
638	uint64_t AllocationSize;	638
639	uint64_t EndOfFile;	639
640	uint32_t FileAttributes;	640
641	} SMB2CloseResponse;	641
642		642
643	#pragma pack(pop)	643
644		644
645	//-----	645
646	// Global State & Helper Functions	646
647	//-----	647
648	static uint64_t gMessageId = 1;	648

649	static uint64_t gSessionId = 0;	649
650	static uint32_t gTreeId = 0;	650
651	static int gSock = -1;	651
652		652
653	static uint64_t gPipeFidPersistent = 0;	653
654	static uint64_t gPipeFidVolatile = 0;	654
655		655
656	/*	656
657	* sendSMB2Request: send an SMB2 header + payload	657
658	*/	658
659	int sendSMB2Request(SMB2Header *hdr, const void *payload, size_t payloadLen) {	659
660	ssize_t sent = send(gSock, hdr, sizeof(SMB2Header), 0);	660
661	if (sent < 0) {	661
662	perror("send header");	662
663	return -1;	663
664	}	664
665	if (payload && payloadLen > 0) {	665
666	sent = send(gSock, payload, payloadLen, 0);	666
667	if (sent < 0) {	667
668	perror("send payload");	668
669	return -1;	669
670	}	670
671	}	671
672	return 0;	672
673	}	673
674		674
675	/*	675
676	* recvSMB2Response: recv an SMB2 header + payload	676
677	*/	677
678	int recvSMB2Response(SMB2Header *outHdr, void *outBuf, size_t bufSize, ssize_t *outPayloadLen) {	678
679	ssize_t recvd = recv(gSock, outHdr, sizeof(SMB2Header), 0);	679
680	if (recvd <= 0) {	680
681	perror("recv SMB2 header");	681
682	return -1;	682
683	}	683
684	if (recvd < (ssize_t)sizeof(SMB2Header)) {	684

685	fprintf(stderr, "Incomplete SMB2 header.\n");	685
686	return -1;	686
687	}	687
688		688
689	// Validate signature	689
690	if (!(outHdr->ProtocolId[0] == 0xFE &&	690
691	outHdr->ProtocolId[1] == 'S' &&	691
692	outHdr->ProtocolId[2] == 'M' &&	692
693	outHdr->ProtocolId[3] == 'B')) {	693
694	fprintf(stderr, "Invalid SMB2 signature.\n");	694
695	return -1;	695
696	}	696
697		697
698	// Non-blocking peek to see if there's more data	698
699	int peekLen = recv(gSock, outBuf, bufSize, MSG_DONTWAIT);	699
700	if (peekLen > 0) {	700
701	int realLen = recv(gSock, outBuf, peekLen, 0);	701
702	if (realLen < 0) {	702
703	perror("recv payload");	703
704	return -1;	704
705	}	705
706	*outPayloadLen = realLen;	706
707	} else {	707
708	*outPayloadLen = 0;	708
709	}	709
710		710
711	return 0;	711
712	}	712
713		713
714	/*	714
715	* buildSMB2Header: fill out common fields	715
716	*/	716
717	void buildSMB2Header(uint16_t command, uint32_t treeId, uint64_t sessionId, SMB2Header *hdrOut) {	717
718	memset(hdrOut, 0, sizeof(SMB2Header));	718
719	hdrOut->ProtocolId[0] = 0xFE;	719
720	hdrOut->ProtocolId[1] = 'S';	720

721	hdrOut->ProtocolId[2] = 'M';	721
722	hdrOut->ProtocolId[3] = 'B';	722
723	hdrOut->StructureSize = 64;	723
724	hdrOut->Command = command;	724
725	hdrOut->Credits = 1; // minimal	725
726	hdrOut->MessageId = gMessageId++;	726
727	hdrOut->TreeId = treeId;	727
728	hdrOut->SessionId = sessionId;	728
729	}	729
730		730
731	//-----	731
732		732
733	// SMB2 NEGOTIATE	733
734	//-----	734
735	int doNegotiate() {	735
736	SMB2Header hdr;	736
737	buildSMB2Header(SMB2_NEGOTIATE, 0, 0, &hdr);	737
738		738
739	SMB2NegotiateRequest req;	739
740	memset(&req, 0, sizeof(req));	740
741	req.StructureSize = 36;	741
742	req.DialectCount = 3;	742
743	uint16_t dialects[3] = {	743
744		744
745	SMB2_DIALECT_0202,	745
746		746
747	SMB2_DIALECT_0210,	747
748		748
749	SMB2_DIALECT_0300	749
750	};	750
751		751
752	// Send header + negotiate request	752
753	if (sendSMB2Request(&hdr, &req, sizeof(req)) < 0) return -1;	753
754	// Followed by the dialect array	754
755	if (send(gSock, dialects, sizeof(dialects), 0) < 0) {	755
756	perror("send dialects");	756

```
757     return -1;
758 }
759
760 // Receive
761 SMB2Header respHdr;
762 unsigned char buf[1024];
763 ssize_t payloadLen;
764 if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;
765
766 if (respHdr.Status != STATUS_SUCCESS) {
767     fprintf(stderr, "Negotiate failed, status=0x%08X\n", respHdr.Status);
768     return -1;
769 }
770 printf("[Client] SMB2 NEGOTIATE OK. payloadLen=%zd\n", payloadLen);
771 return 0;
772 }
773
774 //-----
775 // SMB2 SESSION_SETUP (stub - no real authentication)
776 //-----
777 int doSessionSetup() {
778     SMB2Header hdr;
779     buildSMB2Header(SMB2_SESSION_SETUP, 0, 0, &hdr);
780
781     SMB2SessionSetupRequest ssreq;
782     memset(&ssreq, 0, sizeof(ssreq));
783     ssreq.StructureSize = 25;
784
785     // In real usage, you'd set SecurityBufferOffset/Length and
786     // provide an NTLM/Kerberos token. This is omitted here.
787
788     if (sendSMB2Request(&hdr, &ssreq, sizeof(ssreq)) < 0) return -1;
789
790     SMB2Header respHdr;
791     unsigned char buf[1024];
792     ssize_t payloadLen;
```

793	if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;	793
794		794
795	if (respHdr.Status != STATUS_SUCCESS) {	795
796	fprintf(stderr, "SessionSetup failed, status=0x%08X\n", respHdr.Status);	796
797	return -1;	797
798	}	798
799		799
800	gSessionId = respHdr.SessionId;	800
801	printf("[Client] SMB2 SESSION_SETUP OK. SessionId=0x%llx\n",	801
802	(unsigned long long)gSessionId);	802
803	return 0;	803
804	}	804
805		805
806	//-----	806
807	// SMB2 TREE_CONNECT to \\server\IPC\$	807
808	//-----	808
809	int doTreeConnect(const char *ipcPath) {	809
810	SMB2Header hdr;	810
811	buildSMB2Header(SMB2_TREE_CONNECT, 0, gSessionId, &hdr);	811
812		812
813	SMB2TreeConnectRequest tcreq;	813
814	memset(&tcreq, 0, sizeof(tcreq));	814
815	tcreq.StructureSize = 9;	815
816	tcreq.PathOffset = sizeof(tcreq);	816
817		817
818	uint32_t pathLen = (uint32_t)strlen(ipcPath);	818
819	tcreq.PathLength = pathLen;	819
820		820
821	size_t reqSize = sizeof(tcreq) + pathLen;	821
822	char *reqBuf = (char *)malloc(reqSize);	822
823	if (!reqBuf) {	823
824	fprintf(stderr, "malloc failed\n");	824
825	return -1;	825
826	}	826
827	memcpy(reqBuf, &tcreq, sizeof(tcreq));	827
828	memcpy(reqBuf + sizeof(tcreq), ipcPath, pathLen);	828

829		829
830	if (sendSMB2Request(&hdr, reqBuf, reqSize) < 0) {	830
831	free(reqBuf);	831
832	return -1;	832
833	}	833
834	free(reqBuf);	834
835		835
836	SMB2Header respHdr;	836
837	unsigned char buf[1024];	837
838	ssize_t payloadLen;	838
839	if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) {	839
840	return -1;	840
841	}	841
842		842
843	if (respHdr.Status != STATUS_SUCCESS) {	843
844	fprintf(stderr, "TreeConnect to %s failed, status=0x%08X\n",	844
845	ipcPath, respHdr.Status);	845
846	return -1;	846
847	}	847
848	if (payloadLen < (ssize_t)sizeof(SMB2TreeConnectResponse)) {	848
849	fprintf(stderr, "TreeConnect response too small\n");	849
850	return -1;	850
851	}	851
852		852
853	gTreeld = respHdr.Treeld;	853
854	printf("[Client] TREE_CONNECT to %s OK. Treeld=0x%08X\n", ipcPath, gTreeld);	854
855	return 0;	855
856	}	856
857		857
858	//-----	858
859	// SMB2 CREATE (Open named pipe, e.g. "\\PIPE\\svcctl")	859
860	//-----	860
861	int doOpenPipe(const char *pipeName) {	861
862	SMB2Header hdr;	862
863	buildSMB2Header(SMB2_CREATE, gTreeld, gSessionId, &hdr);	863
864		864

865	SMB2CreateRequest creq;	865
866	memset(&creq, 0, sizeof(creq));	866
867	creq.StructureSize = 57;	867
868	creq.RequestedOplockLevel = 0; // none	868
869	creq.ImpersonationLevel = 2; // SecurityImpersonation	869
870	creq.DesiredAccess = 0x001F01FF; // GENERIC_ALL (over-simplified)	870
871	creq.ShareAccess = 3; // read/write share	871
872	creq.CreateDisposition = 1; // FILE_OPEN	872
873	creq.CreateOptions = 0;	873
874	creq.NameOffset = sizeof(SMB2CreateRequest);	874
875		875
876	// Convert ASCII to a simple UTF-16LE	876
877	uint32_t pipeNameLenBytes = (uint32_t)(strlen(pipeName) * 2);	877
878	creq.NameLength = (uint16_t)pipeNameLenBytes;	878
879		879
880	size_t totalSize = sizeof(creq) + pipeNameLenBytes;	880
881	unsigned char *reqBuf = (unsigned char *)malloc(totalSize);	881
882	if (!reqBuf) {	882
883	fprintf(stderr, "malloc doOpenPipe failed\n");	883
884	return -1;	884
885	}	885
886	memcpy(reqBuf, &creq, sizeof(creq));	886
887		887
888	// ASCII -> UTF-16LE	888
889	unsigned char *pName = reqBuf + sizeof(creq);	889
890	for (size_t i = 0; i < strlen(pipeName); i++) {	890
891	pName[i*2] = (unsigned char)pipeName[i];	891
892	pName[i*2+1] = 0x00;	892
893	}	893
894		894
895	if (sendSMB2Request(&hdr, reqBuf, totalSize) < 0) {	895
896	free(reqBuf);	896
897	return -1;	897
898	}	898
899	free(reqBuf);	899
900		900

```
901     SMB2Header respHdr; 901
902     unsigned char buf[1024]; 902
903     ssize_t payloadLen; 903
904     if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1; 904
905 905
906     if (respHdr.Status != STATUS_SUCCESS) { 906
907         fprintf(stderr, "OpenPipe '%s' failed, status=0x%08X\n", 907
908             pipeName, respHdr.Status); 908
909         return -1; 909
910     } 910
911 911
912     if (payloadLen < (ssize_t)sizeof(SMB2CreateResponse)) { 912
913         fprintf(stderr, "CreateResponse too small.\n"); 913
914         return -1; 914
915     } 915
916     SMB2CreateResponse *cres = (SMB2CreateResponse *)buf; 916
917     gPipeFidPersistent = cres->FileIdPersistent; 917
918     gPipeFidVolatile = cres->FileIdVolatile; 918
919 919
920     printf("[Client] Named pipe '%s' opened OK. FID=(%llx:%llx)\n", 920
921         pipeName, 921
922         (unsigned long long)gPipeFidPersistent, 922
923         (unsigned long long)gPipeFidVolatile); 923
924     return 0; 924
925 } 925
926 926
927 //----- 927
928 // doWritePipe: Send raw bytes into the named pipe 928
929 //----- 929
930 int doWritePipe(const unsigned char *data, size_t dataLen) { 930
931     SMB2Header hdr; 931
932     buildSMB2Header(SMB2_WRITE, gTreeId, gSessionId, &hdr); 932
933 933
934     SMB2WriteRequest wreq; 934
935     memset(&wreq, 0, sizeof(wreq)); 935
936     wreq.StructureSize = 49; 936
```

937	wreq.DataOffset = sizeof(SMB2WriteRequest);	937
938	wreq.Length = (uint32_t)dataLen;	938
939	wreq.FileIdPersistent = gPipeFidPersistent;	939
940	wreq.FileIdVolatile = gPipeFidVolatile;	940
941		941
942	size_t totalSize = sizeof(wreq) + dataLen;	942
943	unsigned char *reqBuf = (unsigned char*)malloc(totalSize);	943
944	if (!reqBuf) {	944
945	fprintf(stderr, "malloc doWritePipe failed\n");	945
946	return -1;	946
947	}	947
948	memcpy(reqBuf, &wreq, sizeof(wreq));	948
949	memcpy(reqBuf + sizeof(wreq), data, dataLen);	949
950		950
951	if (sendSMB2Request(&hdr, reqBuf, totalSize) < 0) {	951
952	free(reqBuf);	952
953	return -1;	953
954	}	954
955	free(reqBuf);	955
956		956
957	// read response	957
958	SMB2Header respHdr;	958
959	unsigned char buf[512];	959
960	ssize_t payloadLen;	960
961	if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;	961
962		962
963	if (respHdr.Status != STATUS_SUCCESS) {	963
964	fprintf(stderr, "WritePipe failed, status=0x%08X\n", respHdr.Status);	964
965	return -1;	965
966	}	966
967	if (payloadLen < (ssize_t)sizeof(SMB2WriteResponse)) {	967
968	fprintf(stderr, "WriteResponse too small\n");	968
969	return -1;	969
970	}	970
971	SMB2WriteResponse *wres = (SMB2WriteResponse *)buf;	971
972	printf("[Client] Wrote %u bytes to pipe.\n", wres->Count);	972

973	return 0;	973
974	}	974
975		975
976	//-----	976
977	// doReadPipe: read back from the pipe	977
978	//-----	978
979	int doReadPipe(unsigned char *outBuf, size_t outBufSize, uint32_t *outBytesRead) {	979
980	SMB2Header hdr;	980
981	buildSMB2Header(SMB2_READ, gTreeld, gSessionId, &hdr);	981
982		982
983	SMB2ReadRequest rreq;	983
984	memset(&rreq, 0, sizeof(rreq));	984
985	rreq.StructureSize = 49;	985
986	rreq.Length = (uint32_t)outBufSize;	986
987	rreq.FileIdPersistent = gPipeFidPersistent;	987
988	rreq.FileIdVolatile = gPipeFidVolatile;	988
989		989
990	if (sendSMB2Request(&hdr, &rreq, sizeof(rreq)) < 0) return -1;	990
991		991
992	SMB2Header respHdr;	992
993	unsigned char buf[2048];	993
994	ssize_t payloadLen;	994
995	if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;	995
996		996
997	if (respHdr.Status != STATUS_SUCCESS) {	997
998	fprintf(stderr, "ReadPipe failed, status=0x%08X\n", respHdr.Status);	998
999	return -1;	999
1000	}	1000
1001	if (payloadLen < (ssize_t)sizeof(SMB2ReadResponse)) {	1001
1002	fprintf(stderr, "ReadResponse too small\n");	1002
1003	return -1;	1003
1004	}	1004
1005	SMB2ReadResponse *rres = (SMB2ReadResponse *)buf;	1005
1006		1006
1007	uint32_t dataLen = rres->DataLength;	1007
1008	if (dataLen > 0) {	1008

1009	uint8_t *dataStart = buf + rres->DataOffset;	1009
1010	// Check for bounds	1010
1011	if (rres->DataOffset + dataLen <= (uint32_t)payloadLen) {	1011
1012	if (dataLen > outBufSize) {	1012
1013	dataLen = (uint32_t)outBufSize; // Truncate	1013
1014	}	1014
1015	memcpy(outBuf, dataStart, dataLen);	1015
1016	} else {	1016
1017	fprintf(stderr, "Data offset/length out of payload bounds!\n");	1017
1018	return -1;	1018
1019	}	1019
1020	}	1020
1021	*outBytesRead = dataLen;	1021
1022	printf("[Client] Read %u bytes from pipe.\n", dataLen);	1022
1023		1023
1024	return 0;	1024
1025	}	1025
1026		1026
1027	//-----	1027
1028	// doDCERPCBind: a partial DCERPC bind request to SVCCTL	1028
1029	//-----	1029
1030	int doDCERPCBind() {	1030
1031	// A typical DCERPC bind to SVCCTL might include:	1031
1032	// - Version/PacketType	1032
1033	// - Interface UUID	1033
1034	// - Transfer syntax, etc.	1034
1035	// This is an oversimplified placeholder.	1035
1036	unsigned char dcerpcBindStub[] = {	1036
1037	0x05, 0x00, // RPC version	1037
1038	0x0B, // bind PDU type	1038
1039	0x10, // flags (little-endian)	1039
1040	0x00, 0x00, 0x00, 0x00, // DCE call ID (placeholder)	1040
1041	// [Interface UUID + version], [transfer syntax], etc...	1041
1042	// This is incomplete for a real DCERPC bind!	1042
1043	};	1043
1044		1044

1045	printf("[Client] Sending partial DCERPC bind stub...\n");	1045
1046	return doWritePipe(dcerpcBindStub, sizeof(dcerpcBindStub));	1046
1047	}	1047
1048		1048
1049	//-----	1049
1050	// doClosePipe: SMB2 Close for the named pipe handle	1050
1051	//-----	1051
1052	int doClosePipe() {	1052
1053	SMB2Header hdr;	1053
1054	buildSMB2Header(SMB2_CLOSE, gTreeld, gSessionId, &hdr);	1054
1055		1055
1056	SMB2CloseRequest creq;	1056
1057	memset(&creq, 0, sizeof(creq));	1057
1058	creq.StructureSize = 24;	1058
1059	creq.Flags = 0; // 0 or 1 for POSTQUERY_ATTR	1059
1060	creq.FileIdPersistent = gPipeFidPersistent;	1060
1061	creq.FileIdVolatile = gPipeFidVolatile;	1061
1062		1062
1063	if (sendSMB2Request(&hdr, &creq, sizeof(creq)) < 0) return -1;	1063
1064		1064
1065	SMB2Header respHdr;	1065
1066	unsigned char buf[512];	1066
1067	ssize_t payloadLen;	1067
1068	if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) {	1068
1069	return -1;	1069
1070	}	1070
1071		1071
1072	if (respHdr.Status != STATUS_SUCCESS) {	1072
1073	fprintf(stderr, "ClosePipe failed, status=0x%08X\n", respHdr.Status);	1073
1074	return -1;	1074
1075	}	1075
1076	printf("[Client] SMB2 Close on pipe handle OK.\n");	1076
1077	return 0;	1077
1078	}	1078
1079		1079
1080	//-----	1080

1081	// main()	1081
1082	//-----	1082
1083	int main(int argc, char *argv[]) {	1083
1084	if (argc < 3) {	1084
1085	fprintf(stderr, "Usage: %s <server_ip> <server_port>\n", argv[0]);	1085
1086	fprintf(stderr, "Example: %s 192.168.1.10 445\n", argv[0]);	1086
1087	return EXIT_FAILURE;	1087
1088	}	1088
1089		1089
1090	const char *serverIp = argv[1];	1090
1091	int port = atoi(argv[2]);	1091
1092		1092
1093	// 1. Create socket	1093
1094	gSock = socket(AF_INET, SOCK_STREAM, 0);	1094
1095	if (gSock < 0) {	1095
1096	perror("socket");	1096
1097	return EXIT_FAILURE;	1097
1098	}	1098
1099		1099
1100	// 2. Connect	1100
1101	struct sockaddr_in serverAddr;	1101
1102	memset(&serverAddr, 0, sizeof(serverAddr));	1102
1103	serverAddr.sin_family = AF_INET;	1103
1104	serverAddr.sin_port = htons(port);	1104
1105	if (inet_pton(AF_INET, serverIp, &serverAddr.sin_addr) <= 0) {	1105
1106	perror("inet_pton");	1106
1107	close(gSock);	1107
1108	return EXIT_FAILURE;	1108
1109	}	1109
1110		1110
1111	if (connect(gSock, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) < 0) {	1111
1112	perror("connect");	1112
1113	close(gSock);	1113
1114	return EXIT_FAILURE;	1114
1115	}	1115
1116	printf("[Client] Connected to %s:%d\n", serverIp, port);	1116

1117		1117
1118	// 3. SMB2 NEGOTIATE	1118
1119	if (doNegotiate() < 0) {	1119
1120	close(gSock);	1120
1121	return EXIT_FAILURE;	1121
1122	}	1122
1123		1123
1124	// 4. SMB2 SESSION_SETUP (stub)	1124
1125	if (doSessionSetup() < 0) {	1125
1126	close(gSock);	1126
1127	return EXIT_FAILURE;	1127
1128	}	1128
1129		1129
1130	// 5. SMB2 TREE_CONNECT to IPC\$	1130
1131	// Construct a UNC path like "\\\192.168.1.10\IPC\$"	1131
1132	char ipcPath[256];	1132
1133	snprintf(ipcPath, sizeof(ipcPath), "\\\%s\IPC\$", serverIp);	1133
1134	if (doTreeConnect(ipcPath) < 0) {	1134
1135	close(gSock);	1135
1136	return EXIT_FAILURE;	1136
1137	}	1137
1138		1138
1139	// 6. SMB2 CREATE for named pipe "\PIPE\svcctl"	1139
1140	if (doOpenPipe("\PIPE\svcctl") < 0) {	1140
1141	close(gSock);	1141
1142	return EXIT_FAILURE;	1142
1143	}	1143
1144		1144
1145	// 7. (Optional) Send a partial DCERPC Bind	1145
1146	if (doDCERPCBind() < 0) {	1146
1147	// Not strictly fatal; you might decide to continue or bail out	1147
1148	fprintf(stderr, "DCERPC bind stub failed.\n");	1148
1149	}	1149
1150		1150
1151	// 8. Attempt a read from the pipe (whatever the server might send back)	1151
1152	unsigned char readBuf[512];	1152

1153	memset(readBuf, 0, sizeof(readBuf));	1153
1154	uint32_t bytesRead = 0;	1154
1155	if (doReadPipe(readBuf, sizeof(readBuf), &bytesRead) < 0) {	1155
1156	fprintf(stderr, "Read from pipe failed.\n");	1156
1157	} else {	1157
1158	if (bytesRead > 0) {	1158
1159	printf("[Client] Pipe response (hex):\n");	1159
1160	for (uint32_t i = 0; i < bytesRead; i++) {	1160
1161	printf("%02X ", readBuf[i]);	1161
1162	}	1162
1163	printf("\n");	1163
1164	} else {	1164
1165	printf("[Client] No data returned from pipe.\n");	1165
1166	}	1166
1167	}	1167
1168		1168
1169	// 9. Close the pipe handle	1169
1170	if (doClosePipe() < 0) {	1170
1171	fprintf(stderr, "Failed to close pipe properly.\n");	1171
1172	}	1172
1173		1173
1174	// 10. Done	1174
1175	close(gSock);	1175
1176	printf("[Client] Done.\n");	1176
1177	return EXIT_SUCCESS;	1177
1178	}	1178
1179		1179
1180		1180
1181		1181
1182		1182
1183		1183
1184		1184
1185		1185
1186		1186
1187		1187
1188		1188

1189		1189
1190		1190
1191		1191
1192		1192
1193		1193
1194		1194
1195		1195
1196		1196
1197		1197
1198	EXHIBIT 3: The “Eternal” family of zero-day exploits developed by the NSA, on the SMBv1 protocol	1198
1199		1199
1200	## A Bit More Detail	1200
1201		1201
1202	1 **The Vulnerability (MS17-010)**	1202
1203	- EternalBlue exploited a memory corruption bug in Microsoft's SMBv1 server (in functions like	1203
1204	`Srv!SrvOs2FeaListToNt` or `Srv!SrvTransaction2Dispatch`).	1204
1205	- By sending specially crafted “trans2” (transaction) packets, the attacker could write arbitrary data past	1205
1206	buffer boundaries in kernel space (in particular, in the `SRV` driver).	1206
1207		1207
1208	2 **Named Pipe vs. Trans2**	1208
1209	- **Named Pipe Exploits (e.g., EternalRomance):** Some SMB exploits from the same leak abused a	1209
1210	named pipe—often `\\pipe\\SRVSVC`—to hold open a file/pipe handle in the SMB server and then	1210
1211	manipulate buffer offsets for code execution.	1211
1212	- **EternalBlue's Approach:** EternalBlue directly abused an out-of-bounds write in the SMBv1 “trans2”	1212
1213	sub-protocol. While SMBv1 does support named pipes, EternalBlue's trigger was not contingent on	1213
1214	obtaining a pipe handle.	1214
1215		1215
1216	3 **Why the Confusion?**	1216
1217	- All these exploits came from the same toolset (Equation Group's FuzzBunch) and target SMB on various	1217
1218	Windows versions.	1218
1219	- EternalBlue, EternalRomance, EternalChampion, and EternalSynergy each had different code paths and	1219
1220	slightly different vulnerabilities, even though they were all SMB-related.	1220
1221		1221
1222	---	1222
1223		1223
1224	### Summary	1224

1225

1226 - **EternalBlue** = Exploits a buffer overflow in SMBv1's "trans2" commands.

1227 - **Does it use a pipe?** No—unlike some sibling exploits (e.g., EternalRomance), it does **not** hinge on

1228 a named pipe handle.

1225

1226

1227

1228

EXHIBIT 1

On 1/30/25, the Plaintiff receives an email from Apple detailing the information request made to the Plaintiff's developer account on 1/8/25, by the Middlesex DA's Office. This date coincided with the Plaintiff filing 27 and 27-1 in Federal Court Case 3:24-cv-06664-JS, the first time ever anyone has won a Section 230 claim vs Twitch interactive.

<https://www.fakeopenai.co/section230>

<https://www.fakeopenai.co/lSAT>

EXHIBIT 2

The Plaintiff is making great progress, and expects to achieve an "Eternal" family of zero-day capabilities on the SMBv2 protocol, within a day or few days.

/*****

* File: smb2_pipe_exec_client.c ss

*

* Demonstrates:

- * 1. Connecting to an SMB2/3 server (TCP 445).
- * 2. Negotiate, Session Setup, Tree Connect to IPC\$.
- * 3. Create/open the named pipe "\\PIPE\\svctl".
- * 4. Partially demonstrate sending a DCERPC bind request to the SVCCTL interface (stub only).
- * 5. Read back any server response.
- * 6. Close the pipe with an SMB2 Close.

*

* Security & Production Warnings:

- * - This remains incomplete demonstration code:
- * - No real auth or signing.
- * - No real DCERPC parse/marshalling logic.
- * - Minimal error handling and no encryption.
- * - Use only in a controlled environment with permission!

*****/

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <unistd.h>
```

```
#include <arpa/inet.h>
```

```
#include <stdint.h>
```

```
#include <errno.h>
```

```
#pragma pack(push, 1)
```

```
//-----
```

```
// SMB2 Header
```

```
//-----
```

EXHIBIT 2

```
typedef struct _SMB2Header {
    unsigned char ProtocolId[4]; // 0xFE 'S' 'M' 'B'
    uint16_t StructureSize; // Always 64 for SMB2
    uint16_t CreditCharge; // Credits requested/charged
    uint32_t Status; // For responses, server sets status
    uint16_t Command; // SMB2 command code
    uint16_t Credits; // Credits granted/requested
    uint32_t Flags; // SMB2 header flags
    uint32_t NextCommand; // Offset to next command in compound
    uint64_t MessageId; // Unique message ID
    uint32_t Reserved; // Usually 0
    uint32_t TreeId; // Tree ID
    uint64_t SessionId; // Session ID
    unsigned char Signature[16]; // For signing (unused here)
} SMB2Header;
```

```
// SMB2 Commands
```

```
#define SMB2_NEGOTIATE 0x0000
#define SMB2_SESSION_SETUP 0x0001
#define SMB2_TREE_CONNECT 0x0003
#define SMB2_CREATE 0x0005
#define SMB2_CLOSE 0x0006
#define SMB2_READ 0x0008
#define SMB2_WRITE 0x0009
```

```
// SMB2 Status Codes (common)
```

```
#define STATUS_SUCCESS 0x00000000
#define STATUS_INVALID_PARAMETER 0xC000000D
#define STATUS_ACCESS_DENIED 0xC0000022
#define STATUS_NOT_SUPPORTED 0xC00000BB
```

```
// SMB2 Dialects
```

```
#define SMB2_DIALECT_0202 0x0202
#define SMB2_DIALECT_0210 0x0210
#define SMB2_DIALECT_0300 0x0300
```

```
//-----
```

EXHIBIT 2

```
// Minimal Structures for Basic SMB2 Ops
```

```
//-----
```

```
/* SMB2 NEGOTIATE */
```

```
typedef struct _SMB2NegotiateRequest {  
    uint16_t StructureSize; // Must be 36  
    uint16_t DialectCount;  
    uint16_t SecurityMode;  
    uint16_t Reserved;  
    uint32_t Capabilities;  
    uint64_t ClientGuid; // Simplified to 8 bytes for demonstration  
    uint32_t NegotiateContextOffset;  
    uint16_t NegotiateContextCount;  
    uint16_t Reserved2;  
    // Then dialect array  
} SMB2NegotiateRequest;
```

```
typedef struct _SMB2NegotiateResponse {  
    uint16_t StructureSize; // Must be 65 in real SMB2  
    uint16_t SecurityMode;  
    uint16_t DialectRevision;  
    uint16_t NegotiateContextCount;  
    uint32_t ServerGuid; // Simplified  
    uint32_t Capabilities;  
    uint32_t MaxTransSize;  
    uint32_t MaxReadSize;  
    uint32_t MaxWriteSize;  
    uint64_t SystemTime;  
    uint64_t ServerStartTime;  
    // etc...  
} SMB2NegotiateResponse;
```

```
/* SMB2 SESSION_SETUP */
```

```
typedef struct _SMB2SessionSetupRequest {  
    uint16_t StructureSize; // Must be 25  
    uint8_t Flags;  
    uint8_t SecurityMode;
```

EXHIBIT 2

```
uint32_t Capabilities;
uint32_t Channel;
uint16_t SecurityBufferOffset;
uint16_t SecurityBufferLength;
// Security buffer follows...
} SMB2SessionSetupRequest;

typedef struct _SMB2SessionSetupResponse {
uint16_t StructureSize; // Must be 9
uint16_t SessionFlags;
uint16_t SecurityBufferOffset;
uint16_t SecurityBufferLength;
// ...
} SMB2SessionSetupResponse;

/* SMB2 TREE_CONNECT */
typedef struct _SMB2TreeConnectRequest {
uint16_t StructureSize; // Must be 9
uint16_t Reserved;
uint32_t PathOffset;
uint32_t PathLength;
// Path follows
} SMB2TreeConnectRequest;

typedef struct _SMB2TreeConnectResponse {
uint16_t StructureSize; // Must be 16
uint8_t ShareType;
uint8_t Reserved;
uint32_t ShareFlags;
uint32_t Capabilities;
uint32_t MaximalAccess;
} SMB2TreeConnectResponse;

/* SMB2 CREATE */
typedef struct _SMB2CreateRequest {
uint16_t StructureSize; // Must be 57
uint8_t SecurityFlags;
```


EXHIBIT 2

```
uint8_t RequestedOplockLevel;
uint32_t ImpersonationLevel;
uint64_t SmbCreateFlags;
uint64_t Reserved;
uint32_t DesiredAccess;
uint32_t FileAttributes;
uint32_t ShareAccess;
uint32_t CreateDisposition;
uint32_t CreateOptions;
uint16_t NameOffset;
uint16_t NameLength;
uint32_t CreateContextsOffset;
uint32_t CreateContextsLength;
// Filename follows...
} SMB2CreateRequest;

typedef struct _SMB2CreateResponse {
uint16_t StructureSize; // Must be 89
uint8_t OplockLevel;
uint8_t Flags;
uint32_t CreateAction;
uint64_t CreationTime;
uint64_t LastAccessTime;
uint64_t LastWriteTime;
uint64_t ChangeTime;
uint64_t AllocationSize;
uint64_t EndOfFile;
uint32_t FileAttributes;
// 16-byte FileId
uint64_t FileIdPersistent;
uint64_t FileIdVolatile;
// optional create contexts
} SMB2CreateResponse;

/* SMB2 WRITE/READ (for the RPC data) */
typedef struct _SMB2WriteRequest {
uint16_t StructureSize; // Must be 49
```

EXHIBIT 2

```
uint16_t DataOffset;
uint32_t Length;
uint64_t Offset;
uint64_t FileIdPersistent;
uint64_t FileIdVolatile;
uint32_t Channel;
uint32_t RemainingBytes;
uint16_t WriteChannelInfoOffset;
uint16_t WriteChannelInfoLength;
uint32_t Flags;
// Then the data
} SMB2WriteRequest;
```

```
typedef struct _SMB2WriteResponse {
uint16_t StructureSize; // Must be 17
uint16_t Reserved;
uint32_t Count;
uint32_t Remaining;
uint16_t WriteChannelInfoOffset;
uint16_t WriteChannelInfoLength;
} SMB2WriteResponse;
```

```
typedef struct _SMB2ReadRequest {
uint16_t StructureSize; // Must be 49
uint8_t Padding;
uint8_t Reserved;
uint32_t Length;
uint64_t Offset;
uint64_t FileIdPersistent;
uint64_t FileIdVolatile;
uint32_t MinimumCount;
uint32_t Channel;
uint32_t RemainingBytes;
uint16_t ReadChannelInfoOffset;
uint16_t ReadChannelInfoLength;
} SMB2ReadRequest;
```

EXHIBIT 2

```
typedef struct _SMB2ReadResponse {
    uint16_t StructureSize; // Must be 17
    uint8_t DataOffset;
    uint8_t Reserved;
    uint32_t DataLength;
    uint32_t DataRemaining;
    uint32_t Reserved2;
    // data follows
} SMB2ReadResponse;
```

```
/* SMB2 CLOSE */
```

```
typedef struct _SMB2CloseRequest {
    uint16_t StructureSize; // Must be 24
    uint16_t Flags;
    uint32_t Reserved;
    uint64_t FileIdPersistent;
    uint64_t FileIdVolatile;
} SMB2CloseRequest;
```

```
typedef struct _SMB2CloseResponse {
    uint16_t StructureSize; // Must be 60
    uint16_t Flags;
    uint32_t Reserved;
    uint64_t CreationTime;
    uint64_t LastAccessTime;
    uint64_t LastWriteTime;
    uint64_t ChangeTime;
    uint64_t AllocationSize;
    uint64_t EndOfFile;
    uint32_t FileAttributes;
} SMB2CloseResponse;
```

```
#pragma pack(pop)
```

```
//-----
```

```
// Global State & Helper Functions
```

```
//-----
```

EXHIBIT 2

```
static uint64_t gMessageId = 1;
```

```
static uint64_t gSessionId = 0;
```

```
static uint32_t gTreeId = 0;
```

```
static int gSock = -1;
```

```
static uint64_t gPipeFidPersistent = 0;
```

```
static uint64_t gPipeFidVolatile = 0;
```

```
/*
```

```
* sendSMB2Request: send an SMB2 header + payload
```

```
*/
```

```
int sendSMB2Request(SMB2Header *hdr, const void *payload, size_t payloadLen) {
```

```
    ssize_t sent = send(gSock, hdr, sizeof(SMB2Header), 0);
```

```
    if (sent < 0) {
```

```
        perror("send header");
```

```
        return -1;
```

```
    }
```

```
    if (payload && payloadLen > 0) {
```

```
        sent = send(gSock, payload, payloadLen, 0);
```

```
        if (sent < 0) {
```

```
            perror("send payload");
```

```
            return -1;
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```

```
/*
```

```
* recvSMB2Response: recv an SMB2 header + payload
```

```
*/
```

```
int recvSMB2Response(SMB2Header *outHdr, void *outBuf, size_t bufSize, ssize_t *outPayloadLen) {
```

```
    ssize_t recvd = recv(gSock, outHdr, sizeof(SMB2Header), 0);
```

```
    if (recvd <= 0) {
```

```
        perror("recv SMB2 header");
```

```
        return -1;
```

```
    }
```

```
    if (recvd < (ssize_t)sizeof(SMB2Header)) {
```

EXHIBIT 2

```
fprintf(stderr, "Incomplete SMB2 header.\n");
return -1;
}
```

```
// Validate signature
if (!(outHdr->ProtocolId[0] == 0xFE &&
outHdr->ProtocolId[1] == 'S' &&
outHdr->ProtocolId[2] == 'M' &&
outHdr->ProtocolId[3] == 'B')) {
fprintf(stderr, "Invalid SMB2 signature.\n");
return -1;
}
```

```
// Non-blocking peek to see if there's more data
int peekLen = recv(gSock, outBuf, bufSize, MSG_DONTWAIT);
if (peekLen > 0) {
int realLen = recv(gSock, outBuf, peekLen, 0);
if (realLen < 0) {
perror("recv payload");
return -1;
}
*outPayloadLen = realLen;
} else {
*outPayloadLen = 0;
}
```

```
return 0;
}
```

```
/*
```

```
* buildSMB2Header: fill out common fields
```

```
*/
```

```
void buildSMB2Header(uint16_t command, uint32_t treeId, uint64_t sessionId, SMB2Header *hdrOut) {
memset(hdrOut, 0, sizeof(SMB2Header));
hdrOut->ProtocolId[0] = 0xFE;
hdrOut->ProtocolId[1] = 'S';
hdrOut->ProtocolId[2] = 'M';
```

EXHIBIT 2

```
hdrOut->ProtocolId[3] = 'B';
hdrOut->StructureSize = 64;
hdrOut->Command = command;
hdrOut->Credits = 1; // minimal
hdrOut->MessageId = gMessageId++;
hdrOut->TreeId = treeld;
hdrOut->SessionId = sessionId;
}

//-----
// SMB2 NEGOTIATE
//-----

int doNegotiate() {
    SMB2Header hdr;
    buildSMB2Header(SMB2_NEGOTIATE, 0, 0, &hdr);

    SMB2NegotiateRequest req;
    memset(&req, 0, sizeof(req));
    req.StructureSize = 36;
    req.DialectCount = 3;
    uint16_t dialects[3] = {
        SMB2_DIALECT_0202,
        SMB2_DIALECT_0210,
        SMB2_DIALECT_0300
    };

    // Send header + negotiate request
    if (sendSMB2Request(&hdr, &req, sizeof(req)) < 0) return -1;
    // Followed by the dialect array
    if (send(gSock, dialects, sizeof(dialects), 0) < 0) {
        perror("send dialects");
        return -1;
    }

    // Receive
    SMB2Header respHdr;
    unsigned char buf[1024];
```

EXHIBIT 2

```

ssize_t payloadLen;
if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;

if (respHdr.Status != STATUS_SUCCESS) {
fprintf(stderr, "Negotiate failed, status=0x%08X\n", respHdr.Status);
return -1;
}
printf("[Client] SMB2 NEGOTIATE OK. payloadLen=%zd\n", payloadLen);
return 0;
}

//-----
// SMB2 SESSION_SETUP (stub - no real authentication)
//-----

int doSessionSetup() {
SMB2Header hdr;
buildSMB2Header(SMB2_SESSION_SETUP, 0, 0, &hdr);

SMB2SessionSetupRequest ssreq;
memset(&ssreq, 0, sizeof(ssreq));
ssreq.StructureSize = 25;

// In real usage, you'd set SecurityBufferOffset/Length and
// provide an NTLM/Kerberos token. This is omitted here.

if (sendSMB2Request(&hdr, &ssreq, sizeof(ssreq)) < 0) return -1;

SMB2Header respHdr;
unsigned char buf[1024];
ssize_t payloadLen;
if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;

if (respHdr.Status != STATUS_SUCCESS) {
fprintf(stderr, "SessionSetup failed, status=0x%08X\n", respHdr.Status);
return -1;
}

```

EXHIBIT 2

```

gSessionId = respHdr.SessionId;
printf("[Client] SMB2 SESSION_SETUP OK. SessionId=0x%llx\n",
(unsigned long long)gSessionId);
return 0;
}

//-----
// SMB2 TREE_CONNECT to \\server\IPC$
//-----

int doTreeConnect(const char *ipcPath) {
SMB2Header hdr;
buildSMB2Header(SMB2_TREE_CONNECT, 0, gSessionId, &hdr);

SMB2TreeConnectRequest tcreq;
memset(&tcreq, 0, sizeof(tcreq));
tcreq.StructureSize = 9;
tcreq.PathOffset = sizeof(tcreq);

uint32_t pathLen = (uint32_t)strlen(ipcPath);
tcreq.PathLength = pathLen;

size_t reqSize = sizeof(tcreq) + pathLen;
char *reqBuf = (char *)malloc(reqSize);
if (!reqBuf) {
fprintf(stderr, "malloc failed\n");
return -1;
}
memcpy(reqBuf, &tcreq, sizeof(tcreq));
memcpy(reqBuf + sizeof(tcreq), ipcPath, pathLen);

if (sendSMB2Request(&hdr, reqBuf, reqSize) < 0) {
free(reqBuf);
return -1;
}
free(reqBuf);

SMB2Header respHdr;

```


EXHIBIT 2

```

unsigned char buf[1024];
ssize_t payloadLen;
if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) {
    return -1;
}

if (respHdr.Status != STATUS_SUCCESS) {
    fprintf(stderr, "TreeConnect to %s failed, status=0x%08X\n",
        ipcPath, respHdr.Status);
    return -1;
}
if (payloadLen < (ssize_t)sizeof(SMB2TreeConnectResponse)) {
    fprintf(stderr, "TreeConnect response too small\n");
    return -1;
}

gTreeld = respHdr.Treeld;
printf("[Client] TREE_CONNECT to %s OK. Treeld=0x%08X\n", ipcPath, gTreeld);
return 0;
}

//-----
// SMB2 CREATE (Open named pipe, e.g. "\\PIPE\\svcctl")
//-----
int doOpenPipe(const char *pipeName) {
    SMB2Header hdr;
    buildSMB2Header(SMB2_CREATE, gTreeld, gSessionId, &hdr);

    SMB2CreateRequest creq;
    memset(&creq, 0, sizeof(creq));
    creq.StructureSize = 57;
    creq.RequestedOplockLevel = 0; // none
    creq.ImpersonationLevel = 2; // SecurityImpersonation
    creq.DesiredAccess = 0x001F01FF; // GENERIC_ALL (over-simplified)
    creq.ShareAccess = 3; // read/write share
    creq.CreateDisposition = 1; // FILE_OPEN
    creq.CreateOptions = 0;

```

EXHIBIT 2

```
creq.NameOffset = sizeof(SMB2CreateRequest);

// Convert ASCII to a simple UTF-16LE
uint32_t pipeNameLenBytes = (uint32_t)(strlen(pipeName) * 2);
creq.NameLength = (uint16_t)pipeNameLenBytes;

size_t totalSize = sizeof(creq) + pipeNameLenBytes;
unsigned char *reqBuf = (unsigned char *)malloc(totalSize);
if (!reqBuf) {
    fprintf(stderr, "malloc doOpenPipe failed\n");
    return -1;
}
memcpy(reqBuf, &creq, sizeof(creq));

// ASCII -> UTF-16LE
unsigned char *pName = reqBuf + sizeof(creq);
for (size_t i = 0; i < strlen(pipeName); i++) {
    pName[i*2] = (unsigned char)pipeName[i];
    pName[i*2+1] = 0x00;
}

if (sendSMB2Request(&hdr, reqBuf, totalSize) < 0) {
    free(reqBuf);
    return -1;
}
free(reqBuf);

SMB2Header respHdr;
unsigned char buf[1024];
ssize_t payloadLen;
if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;

if (respHdr.Status != STATUS_SUCCESS) {
    fprintf(stderr, "OpenPipe '%s' failed, status=0x%08X\n",
        pipeName, respHdr.Status);
    return -1;
}
```

EXHIBIT 2

```

if (payloadLen < (ssize_t)sizeof(SMB2CreateResponse)) {
    fprintf(stderr, "CreateResponse too small.\n");
    return -1;
}
SMB2CreateResponse *cres = (SMB2CreateResponse *)buf;
gPipeFidPersistent = cres->FileIdPersistent;
gPipeFidVolatile = cres->FileIdVolatile;

printf("[Client] Named pipe '%s' opened OK. FID=(%llx:%llx)\n",
    pipeName,
    (unsigned long long)gPipeFidPersistent,
    (unsigned long long)gPipeFidVolatile);
return 0;
}

//-----
// doWritePipe: Send raw bytes into the named pipe
//-----

int doWritePipe(const unsigned char *data, size_t dataLen) {
    SMB2Header hdr;
    buildSMB2Header(SMB2_WRITE, gTreeId, gSessionId, &hdr);

    SMB2WriteRequest wreq;
    memset(&wreq, 0, sizeof(wreq));
    wreq.StructureSize = 49;
    wreq.DataOffset = sizeof(SMB2WriteRequest);
    wreq.Length = (uint32_t)dataLen;
    wreq.FileIdPersistent = gPipeFidPersistent;
    wreq.FileIdVolatile = gPipeFidVolatile;

    size_t totalSize = sizeof(wreq) + dataLen;
    unsigned char *reqBuf = (unsigned char*)malloc(totalSize);
    if (!reqBuf) {
        fprintf(stderr, "malloc doWritePipe failed\n");
        return -1;
    }

```

EXHIBIT 2

```

memcpy(reqBuf, &wreq, sizeof(wreq));
memcpy(reqBuf + sizeof(wreq), data, dataLen);

if (sendSMB2Request(&hdr, reqBuf, totalSize) < 0) {
free(reqBuf);
return -1;
}
free(reqBuf);

// read response
SMB2Header respHdr;
unsigned char buf[512];
ssize_t payloadLen;
if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;

if (respHdr.Status != STATUS_SUCCESS) {
fprintf(stderr, "WritePipe failed, status=0x%08X\n", respHdr.Status);
return -1;
}
if (payloadLen < (ssize_t)sizeof(SMB2WriteResponse)) {
fprintf(stderr, "WriteResponse too small\n");
return -1;
}
SMB2WriteResponse *wres = (SMB2WriteResponse *)buf;
printf("[Client] Wrote %u bytes to pipe.\n", wres->Count);
return 0;
}

//-----
// doReadPipe: read back from the pipe
//-----

int doReadPipe(unsigned char *outBuf, size_t outBufSize, uint32_t *outBytesRead) {
SMB2Header hdr;
buildSMB2Header(SMB2_READ, gTreeId, gSessionId, &hdr);

SMB2ReadRequest rreq;
memset(&rreq, 0, sizeof(rreq));

```

EXHIBIT 2

```

rreq.StructureSize = 49;
rreq.Length = (uint32_t)outBufSize;
rreq.FileIdPersistent = gPipeFidPersistent;
rreq.FileIdVolatile = gPipeFidVolatile;

if (sendSMB2Request(&hdr, &rreq, sizeof(rreq)) < 0) return -1;

SMB2Header respHdr;
unsigned char buf[2048];
ssize_t payloadLen;
if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) return -1;

if (respHdr.Status != STATUS_SUCCESS) {
fprintf(stderr, "ReadPipe failed, status=0x%08X\n", respHdr.Status);
return -1;
}
if (payloadLen < (ssize_t)sizeof(SMB2ReadResponse)) {
fprintf(stderr, "ReadResponse too small\n");
return -1;
}
SMB2ReadResponse *rres = (SMB2ReadResponse *)buf;

uint32_t dataLen = rres->DataLength;
if (dataLen > 0) {
uint8_t *dataStart = buf + rres->DataOffset;
// Check for bounds
if (rres->DataOffset + dataLen <= (uint32_t)payloadLen) {
if (dataLen > outBufSize) {
dataLen = (uint32_t)outBufSize; // Truncate
}
memcpy(outBuf, dataStart, dataLen);
} else {
fprintf(stderr, "Data offset/length out of payload bounds!\n");
return -1;
}
}
*outBytesRead = dataLen;

```

EXHIBIT 2

```
printf("[Client] Read %u bytes from pipe.\n", dataLen);

return 0;
}

//-----
// doDCERPCBind: a partial DCERPC bind request to SVCCTL
//-----
int doDCERPCBind() {
// A typical DCERPC bind to SVCCTL might include:
// - Version/PacketType
// - Interface UUID
// - Transfer syntax, etc.
// This is an oversimplified placeholder.
unsigned char dcerpcBindStub[] = {
0x05, 0x00, // RPC version
0x0B, // bind PDU type
0x10, // flags (little-endian)
0x00, 0x00, 0x00, 0x00, // DCE call ID (placeholder)
// [Interface UUID + version], [transfer syntax], etc...
// This is incomplete for a real DCERPC bind!
};

printf("[Client] Sending partial DCERPC bind stub...\n");
return doWritePipe(dcerpcBindStub, sizeof(dcerpcBindStub));
}

//-----
// doClosePipe: SMB2 Close for the named pipe handle
//-----
int doClosePipe() {
SMB2Header hdr;
buildSMB2Header(SMB2_CLOSE, gTreeId, gSessionId, &hdr);

SMB2CloseRequest creq;
memset(&creq, 0, sizeof(creq));
creq.StructureSize = 24;
```

EXHIBIT 2

```

creq.Flags = 0; // 0 or 1 for POSTQUERY_ATTR
creq.FileIdPersistent = gPipeFidPersistent;
creq.FileIdVolatile = gPipeFidVolatile;

if (sendSMB2Request(&hdr, &creq, sizeof(creq)) < 0) return -1;

SMB2Header respHdr;
unsigned char buf[512];
ssize_t payloadLen;
if (recvSMB2Response(&respHdr, buf, sizeof(buf), &payloadLen) < 0) {
    return -1;
}

if (respHdr.Status != STATUS_SUCCESS) {
    fprintf(stderr, "ClosePipe failed, status=0x%08X\n", respHdr.Status);
    return -1;
}
printf("[Client] SMB2 Close on pipe handle OK.\n");
return 0;
}

//-----
// main()
//-----

int main(int argc, char *argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <server_ip> <server_port>\n", argv[0]);
        fprintf(stderr, "Example: %s 192.168.1.10 445\n", argv[0]);
        return EXIT_FAILURE;
    }

    const char *serverIp = argv[1];
    int port = atoi(argv[2]);

    // 1. Create socket
    gSock = socket(AF_INET, SOCK_STREAM, 0);
    if (gSock < 0) {

```

EXHIBIT 2

```
perror("socket");
return EXIT_FAILURE;
}

// 2. Connect
struct sockaddr_in serverAddr;
memset(&serverAddr, 0, sizeof(serverAddr));
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(port);
if (inet_pton(AF_INET, serverIp, &serverAddr.sin_addr) <= 0) {
perror("inet_pton");
close(gSock);
return EXIT_FAILURE;
}

if (connect(gSock, (struct sockaddr*)&serverAddr, sizeof(serverAddr)) < 0) {
perror("connect");
close(gSock);
return EXIT_FAILURE;
}
printf("[Client] Connected to %s:%d\n", serverIp, port);

// 3. SMB2 NEGOTIATE
if (doNegotiate() < 0) {
close(gSock);
return EXIT_FAILURE;
}

// 4. SMB2 SESSION_SETUP (stub)
if (doSessionSetup() < 0) {
close(gSock);
return EXIT_FAILURE;
}

// 5. SMB2 TREE_CONNECT to IPC$
// Construct a UNC path like "\\192.168.1.10\IPC$"
char ipcPath[256];
```


EXHIBIT 2

```
snprintf(ipcPath, sizeof(ipcPath), "\\\\.\\%s\\IPC$", serverIp);
if (doTreeConnect(ipcPath) < 0) {
    close(gSock);
    return EXIT_FAILURE;
}

// 6. SMB2 CREATE for named pipe "\\PIPE\\svcctl"
if (doOpenPipe("\\PIPE\\svcctl") < 0) {
    close(gSock);
    return EXIT_FAILURE;
}

// 7. (Optional) Send a partial DCERPC Bind
if (doDCERPCBind() < 0) {
    // Not strictly fatal; you might decide to continue or bail out
    fprintf(stderr, "DCERPC bind stub failed.\n");
}

// 8. Attempt a read from the pipe (whatever the server might send back)
unsigned char readBuf[512];
memset(readBuf, 0, sizeof(readBuf));
uint32_t bytesRead = 0;
if (doReadPipe(readBuf, sizeof(readBuf), &bytesRead) < 0) {
    fprintf(stderr, "Read from pipe failed.\n");
} else {
    if (bytesRead > 0) {
        printf("[Client] Pipe response (hex):\n");
        for (uint32_t i = 0; i < bytesRead; i++) {
            printf("%02X ", readBuf[i]);
        }
        printf("\n");
    } else {
        printf("[Client] No data returned from pipe.\n");
    }
}

// 9. Close the pipe handle
```

EXHIBIT 2

```
if (doClosePipe() < 0) {  
    fprintf(stderr, "Failed to close pipe properly.\n");  
}
```

```
// 10. Done  
close(gSock);  
printf("[Client] Done.\n");  
return EXIT_SUCCESS;  
}
```

EXHIBIT 3

The “Eternal” family of zero-day exploits developed by the NSA, on the SMBv1 protocol

A Bit More Detail

1. **The Vulnerability (MS17-010)**

- EternalBlue exploited a memory corruption bug in Microsoft’s SMBv1 server (in functions like `\Srv!SrvOs2FeaListToNt`` or `\Srv!SrvTransaction2Dispatch``).
- By sending specially crafted “trans2” (transaction) packets, the attacker could write arbitrary data past buffer boundaries in kernel space (in particular, in the `\SRV`` driver).

2. **Named Pipe vs. Trans2**

- **Named Pipe Exploits (e.g., EternalRomance):** Some SMB exploits from the same leak abused a named pipe—often `\pipe\SRVSVC``—to hold open a file/pipe handle in the SMB server and then manipulate buffer offsets for code execution.
- **EternalBlue’s Approach:** EternalBlue directly abused an out-of-bounds write in the SMBv1 “trans2” sub-protocol. While SMBv1 does support named pipes, EternalBlue’s trigger was not contingent on obtaining a pipe handle.

3. **Why the Confusion?**

- All these exploits came from the same toolset (Equation Group’s FuzzBunch) and target SMB on various Windows versions.
- EternalBlue, EternalRomance, EternalChampion, and EternalSynergy each had different code paths and slightly different vulnerabilities, even though they were all SMB-related.

Summary

- **EternalBlue** = Exploits a buffer overflow in SMBv1’s “trans2” commands.
- **Does it use a pipe?** No—unlike some sibling exploits (e.g., EternalRomance), it does **not** hinge on a named pipe handle.