SKATY

Scapy → Kotlin

```yaml
%YAML 1.2
---
project:
  name: Skaty
  description: Kotlin port for Scapy library
  author:
    name: Yoav Sternberg
    id: 315144345
    class: 12'4
    school: Ohel Shem
    city: Ramat Gan
  technologies:
    - Python # Scapy is a python library
    - Kotlin # Code written in Kotlin
    - JVM    # The platform. Uses pcap4j for native binding
    - Food   # Hamburgers, Pizzas, Fruits and more…
  repo: https://github.com/yoavst/Skaty
  teacher: Tomer Talgam
```

INDEX OF /

מבוא

סקאפי (באנגלית: Scapy) תוכנית קוד פתוח, כתובה בשפת פייתון ומספקת ניתוח תעבורת רשת, שליחה ובדיקת חבילות מידע בפרוטוקולים שונים ובעלת יכולות לתקיפת רשתות באמצעות כתיבת סקריפטים. סקאפי היא ספרייה מעולה, אך יש לה מספר חסרונות.

ראשית, היא כתובה בשפה דינאמית. עקב כך, היכולת להבין מה הפרמטרים שהפונקציות מקבלות, ומאיזה סוג הם, נפגעת. עקב כך, התאימות עם ה-IDE לא כל כך טובה. שנית, לסקאפי יש כל מיני באגים מוזרים שמקשים על השימוש בה.

הפרויקט בא לפתור את הבעיה הראשונה, בכך שהוא מפותח בשפה שהיא Type safe. מטרתו היא לבנות POC של ספרייה אשר התחביר שלה דומה ככל שניתן לסקאפי, אולם בלי "קסמים", אלא באמצעות OOP.

השפה שנבחרה היא Kotlin, השפה "החדשה" של חברת Jetbrains (היוצרת של סביבות הפיתוח PyCharm ו Intellij IDEA). הפיצ'רים בשפה שבזכותם בחרתי בשפה לפרויקט הם:

- Lambda support – הפעולות המובנות על Collections והתמיכה המובנת ב-Lambda functions מאפשרות עבודה עם רצף של פקטות שיגיעו מפונקציית ה-Sniff בצורה שנוחה בהרבה מ-Scapy.
- Default values – האפשרות לתת ערך ברירת מחדל לפרמטרים שלא מולאו על ידי המשתמש מאפשרת ממשק שיראה כמו סקאפי במקום ממשק מבוסס Builder pattern, כמו שנוקטות ספריות כגון Pacp4j
- Property reference – מאפשר דרך "בטוחה" (באמצעות Reflections אומנם, אבל בטוחה) לאפס את הערך של משתנה לערך ברירת המחדל שלו (פרטים בגוף העבודה)
- Companion Object – מאפשר לקשר אובייקט שיהיה אחראי על כל העבודה עם הפרוטוקול, עם הפרוטוקול עצמו.

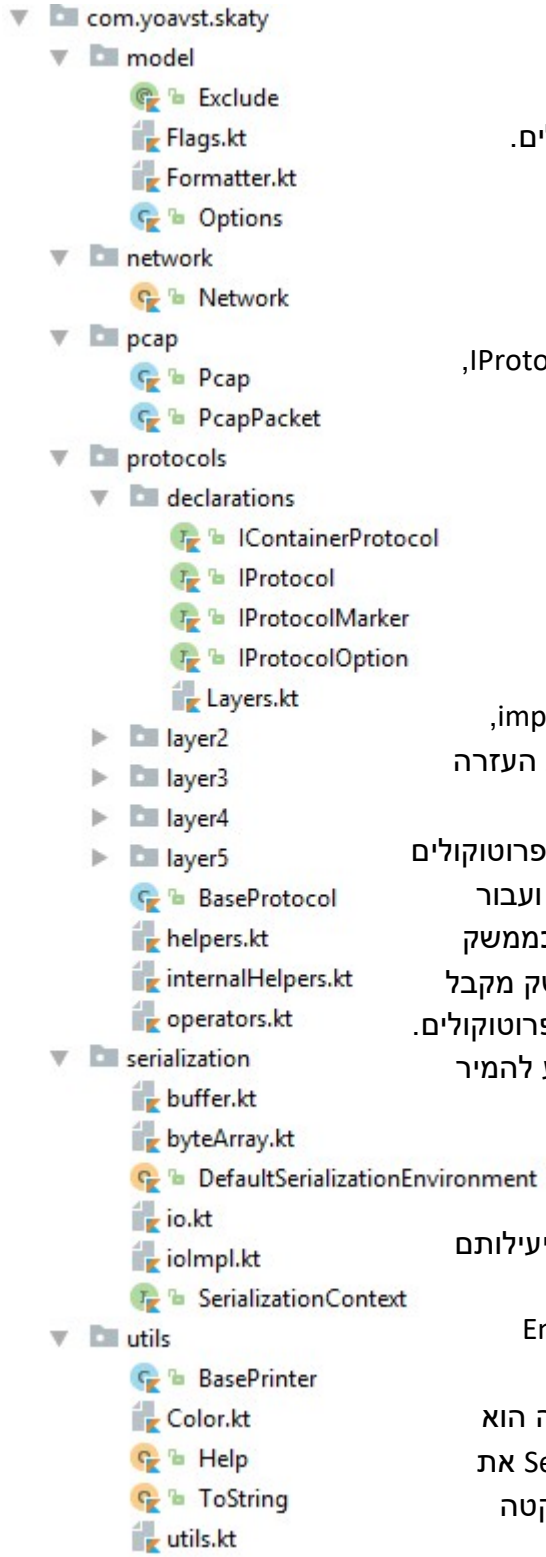
בחרתי בפרויקט כי אני מעדיף שפות שהם Type Safe על שפות שהן לא, וכמות הזמן שביזבזתי על פרמטרים לא נכונים ל-Scapy היא OVER 9000.

כלים

השפה בה השתמשתי, כפי שציינתי במבוא, היא Kotlin, בגרסה 1.1.1, בלי שימוש ב- Experimental Coroutines support. סביבת הפיתוח היא Intellij IDEA בגרסת 2017.1.

הספרייה מתבססת על ספריית Pcap4j, אשר משתמש כ-bridge לספריית libpcap. מלבד קבלת פקטות בצורה בינארית, ושליחת פקטות בצורה בינארית, אין שימוש בספרייה.

מבנה הפרויקט



הפרויקט מורכב מארבעה חלקים:

1. POJO של פרוטוקולים (protocol) – ההגדרת מחלקות הפרוטוקולים.
בפרויקט עצמו נתונות ההגדרות של:
Ethernet, IP, TCP, UDP, Raw.
כל מחלקה של פרוטוקול צריכה לממש את IProtocol, או
את IContainerProtocol במידה והיא יכולה להכיל פרוטוקולים
אחרים. לכל IProtocol יש מצביע למופע שמממש את IProtocolMarker,
כאשר מומלץ שהוא יהיה ה Companion object של המחלקה.
נוסף על כך, מומלץ שפרוטוקול יצהיר את השכבה שלו באמצעות
הרחבתו מהממשק LayerN כאשר N הוא מספר בין 1 ל5.
ההגדרה של IProtocol היא מינימלסטית, ולכן כל הפעולות על
פרוטוקול מוגדרות כ Extension methods (כמו פעולת החילוק
שמשמשת להוספת Payload או פעולת [] שמאפשרת לגשת
לPayload מסוג מסוים), תחת הקובץ operators.kt.
זאת ועוד, הואיל וזאת המחלקה אותה הספרייה מייצאת עבור import,
ישנם עזרים כללים שנמצאים תחת הקובץ helpers.kt כגון פעולות העזרה
()ls ו ()lsOption.
2. סריאליזציה ודסריאליזציה (Serialization) – כתיבה וקריאה של הפרוטוקולים
למידע בינארי. כל פרוטוקול צריך לממש עבור מופע את הכתיבה, ועבור
מרקר את הקריאה. הספרייה מגדירה את SerializationContext כממשק
שבאמצעותו ניתן לקרוא ולכתוב את הפרוטוקולים לבינארי. הממשק מקבל
פרוטוקול כללי וממיר אותו לבינארי, או בינארי כללי וממיר אותו לפרוטוקולים.
את "הקסם" שבאמצעותו DefaultSerializationEnvironment יודע להמיר
לפרוטוקול הנכון, אסביר בהמשך.
מלבד זאת, יש בחבילה את ההגדרות של SimpleWriter
וSimpleReader, שהם הממשקים שבאמצעותם הספרייה
עוטפת את הקריאה והכתיבה לבינארי כדי שניתן יהיה לשפר את יעילותם
בעתיד. המימוש הדיפולטיבי בחבילה משתמש במערך של בתים
עבור קורא פשוט, וב-ByteBuffer עבור קורא שתומך בשינוי Endian
ועבור כותב.
3. Pcap – מימוש התמיכה בקריאת קובץ Pcap. המימוש של הקריאה הוא
כמימוש של פרוטוקול רגיל. ה-pcap מעביר ל-SerializationContext את
הפקטות שיש לו, ומקבל חזרה את הפרוטוקולים. הוא עוטף כל פקטה
במידע שיש לו בקובץ, שהוא הזמן שבו הוסנפה וגודלה.
4. רשת (network) – מימוש נאיבי במקצת של שליחת פקטות והסנפת פקטות.
כרגע מתבסס על Pcap4j עבור העטיפה של libpcap.

דוגמאות והרחבה

```
init("192.168.1.3") // Currently initializing with current IP address is required

val p = Ether(dst = mac("11-22-33-44-55-66")) /
        IP(dst = ip("192.168.1.1"), tos = 53.ub, options = optionsOf(MTUProb(22.us))) /
        TCP(dport = 80.us, sport = 1200.us, flags = flagsOf(SYN, ACK)) /
        "Hello world"

ls(IP)

// work with properties
p.dst = mac("AA-BB-CC-DD-EE-FF")
del(p::dst)
println(p)

val p2 = packet.copy(type = 1.us) // Shallow copy

// get a layer
val tcp = p[TCP]
println(tcp.dport)
println(UDP in packet)

// send packet
sendp(p)

// sniff
val cap = sniff(timeout = 50).filter { TCP in it && it[TCP].dport == 1200.us }.take(10).toList()
cap.forEach(::println)
```

בעמוד הבא ישנו מימוש של פרוטוקול הצאט מתרגיל 12.6 שבספר הפייטון של גבהים.
מספר הערות לגביו:

- רק פרוטוקולים שאחד מהפרמטרים שלהם הוא ByteArray או מערך כלשהו, דורשים מימוש מחדש של equals ו hashcode. אחרת, הוספת המילה data לפני המחלקה אמורה ליצור לבד.
- מנגון הסריאליזציה והשלבים שלו יוסברו בהמשך
- כדי שהפרוטוקול יזוהה על ידי DefaultSerializationEnvironment, יש לעשות לו binding.

```
DefaultSerializationEnvironment.bind(TCP::dport, 3232, ChatProto)
DefaultSerializationEnvironment.bind(TCP::sport, 3232, ChatProto)
```

```
data class ChatProto(var usernameLength: Ubyte = 0.ub, var username: String = "",
                     var command: Byte = 0, var parameters: ByteArray = ByteArray(0),
                     override var parent: IProtocol<*>? = null) : IProtocol<ChatProto> {
    override fun write(writer: SimpleWriter, stage: Stage) {
        when (stage) {
            Stage.Data -> {
                val bytes = username.toByteArray(charset = Charsets.UTF_8)
                if (usernameLength == 0.ub)
                    usernameLength = bytes.size.ub
                writer.writeUbyte(usernameLength)
                writer.writeByteArray(bytes, length = usernameLength.toInt())
                writer.writeByte(command)
                writer.writeByteArray(parameters)
            }
            Stage.Length -> {
                writer.skip(headerSize())

            }
            Stage.Checksum -> {
                writer.index -= headerSize()
            }
        }
    }

    override fun toString(): String = ToString.generate(this)
    override val marker get() = Companion
    override fun headerSize(): Int = 2 + username.toByteArray().size + parameters.size
    /* If protocol usage ByteArray, it has to reimplement equals(any) and hashcode() functions */
    override fun equals(other: Any?): Boolean {
        return other === this || (other is TestProtocol && other.usernameLength == usernameLength &&
other.username == username &&
                other.command == command && parameters.contentEquals(other.parameters ?: byteArrayOf()))
    }

    override fun hashCode(): Int {
        var result = usernameLength.toInt()
        result = 31 * result + username.hashCode()
        result = 31 * result + command
        result = 31 * result + parameters.contentHashCode()
        return result
    }

    companion object : IProtocolMarker<ChatProto> {
        override val name: String get() = "12.6 Chat"
        override fun isProtocol(protocol: IProtocol<*>): Boolean = protocol is ChatProto
        override val defaultValue: ChatProto = ChatProto()

        override fun of(reader: SimpleReader, serializationContext: SerializationContext): ChatProto? = try {
            val usernameLength = reader.readUbyte()
            val username = String(reader.readByteArray(usernameLength.toInt()), charset = Charsets.UTF_8)
            val command = reader.readByte()
            val parameters = reader.readAsByteArray() // read until the end
            ChatProto(usernameLength, username, command, parameters)
        } catch (e: Exception) {
            println("Failed to serialize to 12.6 chat protocol")
            e.printStackTrace()
            null
        }
    }
}
```

קוד

הקוד המלא נמצא בקישור https://github.com/yoavst/Skaty

```
interface IProtocol<K : IProtocol<K>> {
    /**
     * The marker is the static extension for the protocol.
     */
    @Exclude
    val marker: IProtocolMarker<K>

    @Exclude
    var parent: IProtocol<*>?

    /**
     * Returns header size in bytes
     */
    fun headerSize(): Int

    /**
     * Write the protocol's data to the [writer] for the given [stage].
     *
     * [Stage.Data]: in: start :: out: end
     * [Stage.Length]: in: start :: out: end
     * [Stage.Checksum]: in: end :: out: start
     *
     */
    fun write(writer: SimpleWriter, stage: Stage)
}
```

```
/**
 * A protocol that may contain another protocol as its payload.
 */
interface IContainerProtocol<K : IContainerProtocol<K>> : IProtocol<K> {
    /**
     * The payload of the protocol.
     */
    @Exclude
    var payload: IProtocol<*>?
}
```

```kotlin
/**
 * Static extension for the protocol [K].
 */
interface IProtocolMarker<K : IProtocol<K>> {
    /**
     * Return whether or not [protocol] is [K]
     *
     * suggested implementation: `protocol is K`
     */
    fun isProtocol(protocol: IProtocol<*>): Boolean

    /**
     * The name of the protocol, used for help
     */
    val name: String

    /**
     * Immutable version of the defaultValue value of the protocol.
     * Changing fields inside the defaultValue value has undefined behavior
     */
    val defaultValue: K

    /**
     * Try to parse the data into the protocol [K]
     */
    fun of(reader: SimpleReader, serializationContext: SerializationContext = DefaultSerializationEnvironment): K?

    operator fun invoke(reader: SimpleReader): K = of(reader) ?: throw IllegalArgumentException("reader does not represent $name protocol.")
}
```

```kotlin
operator fun <K : IContainerProtocol<*>> K.div(protocol: IProtocol<*>): K {
    val payload = this.payload
    if (payload == null) this.payload = protocol
    else if (payload !is IContainerProtocol<*>) throw IllegalArgumentException("cannot put payload to non container layer. Layer is: ${payload.marker.name}")
    else payload / protocol
    return this
}

operator fun <K : IContainerProtocol<*>> K.div(load: String): K = div(Raw(load.toByteArray()))
operator fun IProtocol<*>?.contains(protocol: IProtocolMarker<*>): Boolean {
    if (this == null) return false
    if (protocol.isProtocol(this)) return true
    else if (this !is IContainerProtocol<*>) return false
    return protocol in payload
}

@Suppress("UNCHECKED_CAST")
operator fun <K : IProtocol<K>> IProtocol<*>?.get(protocol: IProtocolMarker<K>): K {
    return getOrNull(protocol) ?: throw IllegalArgumentException("Protocol ${protocol.name} is not found")
}

@Suppress("UNCHECKED_CAST")
fun <K : IProtocol<K>> IProtocol<*>?.getOrNull(protocol: IProtocolMarker<K>): K? {
    if (this == null) return null
    if (protocol.isProtocol(this)) return this as K
    else if (this !is IContainerProtocol<*>) return null
    else return payload[protocol]
}

fun <T> del(property: KMutableProperty0<T>) {
    property.set(property.default())
}

@Suppress("UNCHECKED_CAST")
fun <T> KProperty<T>.default(): T {
    return getter.javaMethod!!.invoke(((this as? CallableReference)?.boundReceiver as? IProtocol<*>)?.marker?.defaultValue) as T
}

@Suppress("UNCHECKED_CAST")
fun <T> KProperty<Any?>.default(instance: IProtocol<*>): T {
    return getter.javaMethod!!.invoke(instance.marker.defaultValue) as T
}
```

```kotlin
interface SimpleWriter : Closeable {
    val size: Int
    var index: Int
    val maxIndex: Int

    fun writeBool(value: Boolean)
    fun writeByte(value: Byte)
    fun writeShort(value: Short)
    fun writeInt(value: Int)
    fun writeLong(value: Long)
    fun writeByteArray(value: ByteArray, offset: Int = 0, length: Int = -1)

    fun array(): ByteArray
}

fun SimpleWriter.writeUbyte(value: Ubyte) = writeByte(value.toByte())
fun SimpleWriter.writeUshort(value: Ushort) = writeShort(value.toShort())
fun SimpleWriter.writeUint(value: Uint) = writeInt(value.toInt())
fun SimpleWriter.writeUlong(value: Ulong) = writeLong(value.toLong())
fun SimpleWriter.writeString(value: String, charset: Charset = Charsets.UTF_8) = writeByteArray(value.toByteArray(charset))
fun SimpleWriter.skip(bytes: Int): Int {
    if (bytes > 0) {
        if (bytes + index ≥ size) {
            val d = size - index - 1
            index = size - 1
            return d
        }

        index += bytes
        return bytes
    } else return 0
}
```

```kotlin
interface SimpleReader : Closeable {
    fun readBool(): Boolean
    fun readByte(): Byte
    fun readShort(): Short
    fun readInt(): Int
    fun readLong(): Long
    /**
     * if [length] == -1, read all the buffer
     */
    fun readByteArray(length: Int): ByteArray

    fun skip(bytes: Int): Int
    fun hasMore(): Boolean
}

interface EndianSimpleReader : SimpleReader {
    fun bigEndian()
    fun littleEndian()
}

fun SimpleReader.readUbyte() = readByte().ub
fun SimpleReader.readUshort() = readShort().us
fun SimpleReader.readUint() = readInt().ui
fun SimpleReader.readUlong() = readLong().ul
fun SimpleReader.readString(length: Int, charset: Charset): String = String(readByteArray(length), charset)
fun SimpleReader.readAsByteArray(): ByteArray = readByteArray(-1)
```

```kotlin
interface SerializationContext {
    fun deserialize(reader: SimpleReader, parent: IProtocol<*>? = null): IProtocol<*>?
    fun serialize(protocol: IProtocol<*>, writer: SimpleWriter, stage: Stage)

    enum class Stage {
        Data, Length, Checksum;

        fun next(): Stage? = if (this == Data) Length else if (this == Length) Checksum else null
    }
}
```

```kotlin
fun mac(address: String): Ether.MAC = Ether.MAC(address.toMacAddress())

fun mac(address: ByteArray): Ether.MAC {
    val int = address.readUint()
    val short = address.readUshort(index = 4)
    return Ether.MAC((int.toUlong() shl 16) or short.toUlong())
}

fun ip(address: String): IP.Address {
    return IP.Address(address.toIpAddress())
}

fun ip(address: ByteArray) : IP.Address {
    return IP.Address(address.readUint())
}

fun ip(address: InetAddress): IP.Address {
    return ip(address.address)
}

fun pcapOf(path: String): Pcap? = pcapOf(File(path))

fun pcapOf(file: File): Pcap? = Pcap.of(BufferSimpleReader(file.readBytes()))

fun <K : IProtocol<K>> IProtocolMarker<K>.of(raw: Raw, serializationContext:
SerializationContext = DefaultSerializationEnvironment): K? {
    return of(ByteArraySimpleReader(raw.load), serializationContext)
}

fun ls(protocol: IProtocolMarker<*>) = println(Help.generate(protocol))

fun IProtocol<*>.toByteArray(serializationContext: SerializationContext =
DefaultSerializationEnvironment): ByteArray {
    val array = ByteArray(65535)
    val writer = ByteArraySimpleWriter(array)
    serializationContext.serialize(this, writer, SerializationContext.Stage.Data)
    return array.sliceArray(0 until writer.maxIndex)
}

@Suppress("UNCHECKED_CAST")
inline fun <reified K: IProtocolOption<in K>> lsOption() =
println(Help.optionGenerate(K::class as KClass<IProtocolOption<*>>))

// kotlin type alias bug, uses original class
fun <K : IProtocol<K>> optionsOf(vararg options: K) = Options(options.toMutableList())
fun <K : IProtocol<K>> emptyOptions() = optionsOf<K>()
```

```
object DefaultSerializationEnvironment : SerializationContext {
    private val bindingProperties: MutableMap<KProperty<*>, MutableMap<Any?,
IProtocolMarker<*>>> = mutableMapOf()
    fun bind(property: KProperty<*>, value: Any?, protocol:
IProtocolMarker<*>) {
        val valuesMap = bindingProperties[property]
        if (valuesMap == null) {
            bindingProperties[property] = mutableMapOf(value to protocol)
        } else {
            if (value !in valuesMap) {
                valuesMap[value] = protocol
            }
        }
    }

    init {
        bind(Pcap::dataLink, 1.ui, Ether)

        bind(Ether::type, 2048.us, IP)

        bind(IP::proto, 4.ub, IP)
        bind(IP::proto, 6.ub, TCP)
        bind(IP::proto, 17.ub, UDP)
    }


    override fun deserialize(reader: SimpleReader, parent: IProtocol<*>?):
IProtocol<*>? {
        if (reader.hasMore()) {
            if (parent == null) {
                return Raw(reader.readAsByteArray())
            } else {
                val properties = parent::class.memberProperties
                for (property in bindingProperties.keys) {
                    if (property in properties) {
                        val valuesMap = bindingProperties[property]!!
                        val protocol = valuesMap[property.getter.call(parent)]
                        if (protocol != null) {
                            val result = protocol.of(reader, this)
                            if (result != null) return result
                        }
                    }
                }
                return Raw(reader.readAsByteArray())
            }
        } else return null
    }

    override fun serialize(protocol: IProtocol<*>, writer: SimpleWriter,
stage: Stage) {
        if (stage == Stage.Checksum) {
            if (protocol is IContainerProtocol<*>)
                protocol.payload?.let { serialize(it, writer, stage) }
            protocol.write(writer, stage)
```

```
        } else {
            @Suppress("NAME_SHADOWING")
            var p: IProtocol<*>? = protocol
            while (true) {
                if (p == null)
                    break

                p.write(writer, stage)
                p = (p as? IContainerProtocol<*>)?.payload
            }
            val next = stage.next()
            if (next != Stage.Checksum)
                writer.index = 0
            if (next != null)
                serialize(protocol, writer, next)
        }
    }
}
```

```
object Network : Closeable {
    private lateinit var readHandle: PcapHandle
    private lateinit var sendHandle: PcapHandle
    private var address: InetAddress =
InetAddress.getByAddress(ip("127.0.0.1").toByteArray())

    internal var index: AtomicLong = AtomicLong(0)
    internal var currentPacket: Pair<IProtocol<*>, Long> = Raw(ByteArray(0))
to index.get()

    val macAddress: Ether.MAC
        get() =
mac(NetworkInterface.getByInetAddress(address).hardwareAddress)

    val ipAddress: IP.Address
        get() = ip(address.address)

    fun init(ip: String) {
        address = InetAddress.getByName(ip)
        val nif = Pcaps.getDevByAddress(address)
        sendHandle = nif.openLive(65536, PromiscuousMode.NONPROMISCUOUS, 0)
        Runtime.getRuntime().addShutdownHook(Thread {
            close()
        })
        thread(start = true, isDaemon = false) {
            readHandle = nif.openLive(65536, PromiscuousMode.NONPROMISCUOUS,
0)
            while (readHandle.isOpen) {
                val raw = readHandle.nextRawPacket
                if (raw != null) {
                    val packet = Ether.of(ByteArraySimpleReader(raw))
                    if (packet != null)
                        currentPacket = packet to index.incrementAndGet()
                }
            }
        }
    }

    override fun close() {
        readHandle.close()
        sendHandle.close()
    }

    fun sniff(timeout: Long = Long.MAX_VALUE): Sequence<IProtocol<*>> {
        val startTime: Long = System.currentTimeMillis()
        var iteratorIndex: Long = index.get()
        return generateSequence {
            while (true) {
                val snifferIndex = index.get()
                if (snifferIndex > 0 && snifferIndex > iteratorIndex) break
                else if (System.currentTimeMillis() - startTime >= timeout)
                    return@generateSequence null
            }
            val (packet, newIndex) = currentPacket
```

```
            iteratorIndex = newIndex
            packet
        }
    }

    fun sendp(packet: Ether) {
        sendHandle.sendPacket(packet.toByteArray())
    }

    fun send(packet: IP) {
        sendHandle.sendPacket((Ether() / packet).toByteArray())
    }
}
```

```
data class Pcap(var major: Ushort = 2.us, var minor: Ushort = 4.us, var
thisZone: Int = 0, var sigfigs: Uint = 0.ui, var maxSnapLen: Uint = 65536.ui,
                var dataLink: Uint = 1.ui, val packets:
MutableList<PcapPacket> = mutableListOf()): IProtocol<Pcap> {
    override val marker get() = Companion
    override fun headerSize(): Int = 24
    override var parent: IProtocol<*>?
        get() = null
        set(value) {}

    override fun toString(): String {
        val counts = mutableMapOf<String, Int>()
        packets.asSequence()
                .map { it().markerName() }
                .forEach { counts[it] = counts.getOrDefault(it, 0) + 1 }

        return "<$name: ${counts.toList().joinToString(separator = " ") {
(name, count) -> "$name:$count" }} />"
    }

    private fun IProtocol<*>.markerName(): String {
        if (this is IContainerProtocol<*> && payload != null &&
!Raw.isProtocol(payload!!))
            return payload!!.markerName()
        else
            return marker.name
    }

    override fun write(writer: SimpleWriter, stage:
SerializationContext.Stage) {
        throw IllegalStateException("Should not call this method on pcap
file")
    }

    operator fun iterator(): Iterator<PcapPacket> = packets.iterator()

    companion object : IProtocolMarker<Pcap>, KLogging() {
        override val name: String = "Pcap file"
```

```
        override fun isProtocol(protocol: IProtocol<*>): Boolean = protocol is
Pcap
        override val defaultValue: Pcap = Pcap()

        override fun of(reader: SimpleReader, serializationContext:
SerializationContext): Pcap? = try {
            if (reader is EndianSimpleReader) {
                reader.bigEndian()
                var magicNumber = reader.readUint().toLong()
                if (magicNumber == 0xd4c3b2a1) {
                    reader.littleEndian()
                    magicNumber = 0xa1b2c3d4
                }

                if (magicNumber != 0xa1b2c3d4)
                    throw IllegalArgumentException("source is not pcap")

                val major = reader.readUshort()
                val minor = reader.readUshort()
                require(major == 2.us && minor == 4.us) { "pcap version is not
supported. Supported: 2.4" }


                val thisZone = reader.readInt() /* GMT to local correction */
                val sigfigs = reader.readUint()
                val maxSnapLen = reader.readUint()
                val dataLink = reader.readUint()

                val packets = LinkedList<PcapPacket>()
                val pcap = Pcap(major, minor, thisZone, sigfigs, maxSnapLen,
dataLink, packets)
                while (reader.hasMore()) {
                    val epoch = reader.readUint()
                    val micros = reader.readUint()
                    val packetFileSize = reader.readUint() /* number of bytes
of packet saved in file */
                    val originalSize = reader.readUint() /* actual length of
packet */
                    if (packetFileSize != originalSize)
                        logger.warn { "mismatch! original size: $originalSize,
stored: $packetFileSize" }

                    val data = reader.readByteArray(packetFileSize.toInt())
                    val result =
serializationContext.deserialize(ByteArraySimpleReader(data), pcap) ?: throw
Exception("Had a problem reading pcap")
                    packets += PcapPacket(result, epoch, micros,
packetFileSize, originalSize)
                }
                pcap
            } else {
                throw IllegalArgumentException("reader must be endian reader")
            }
        } catch (e: Exception) {
```

```
            logger.error(e) { "Failed to parse pcap file" }
            null
        }
    }
}
```

```
data class Raw(@property:Formatted(Companion::class) var load: ByteArray =
byteArrayOf(), override var parent: IProtocol<*>? = null) : IProtocol<Raw> {
    override fun toString(): String = ToString.generate(this)
    override val marker get() = Companion
    override fun headerSize(): Int = 0

    operator fun contains(text: String) = text in String(load)

    override fun write(writer: SimpleWriter, stage:
SerializationContext.Stage) {
        when (stage) {
            Stage.Data -> {
                writer.writeByteArray(load)
            }
            Stage.Length -> {
                writer.skip(load.size)
            }
            Stage.Checksum -> {
                writer.index -= load.size
            }
        }

    }

    override fun equals(other: Any?): Boolean = (other as?
Raw)?.load?.contentEquals(load) ?: false

    override fun hashCode(): Int = load.contentHashCode()

    companion object : IProtocolMarker<Raw>, Formatter<ByteArray> {
        override val name: String get() = "Raw"
        override fun isProtocol(protocol: IProtocol<*>): Boolean = protocol is
Raw
        override val defaultValue: Raw = Raw()

        override fun of(reader: SimpleReader, serializationContext:
SerializationContext): Raw? = Raw(reader.readAsByteArray())

        override fun format(value: ByteArray?): String = "\"${(value?.let {
String(it, Charsets.UTF_8) } ?: "").replace("\r\n", "\\n").replace("\n",
"\\n")}\""
    }
}
```